Finite difference methods for vibration problems

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Sep 11, 2015

Contents

Vibration problems lead to differential equations with solutions that oscillate in time, typically in a damped or undamped sinusoidal fashion. Such solutions put certain demands on the numerical methods compared to other phenomena whose solutions are monotone or very smooth. Both the frequency and amplitude of the oscillations need to be accurately handled by the numerical schemes. Most of the reasoning and specific building blocks introduced in the forthcoming text can be reused to construct sound methods for partial differential equations of wave nature in multiple spatial dimensions.

1 Finite difference discretization

Many of the numerical challenges faced when computing oscillatory solutions to ODEs and PDEs can be captured by the very simple ODE $u'' + u = 0$. This ODE is thus chosen as our starting point for method development, implementation, and analysis.

1.1 A basic model for vibrations

A system that vibrates without damping and external forcing can be described by the ODE problem

$$u'' + \omega^2 u = 0, \quad u(0) = I,\ u'(0) = 0,\ t \in (0, T]\,. \tag{1}$$

Here, ω and I are given constants. The exact solution of (1) is

$$u(t) = I\cos(\omega t)\,. \tag{2}$$

That is, u oscillates with constant amplitude I and angular frequency ω. The corresponding period of oscillations (i.e., the time between two neighboring peaks in the cosine function) is $P = 2\pi/\omega$. The number of periods per second is $f = \omega/(2\pi)$ and measured in the unit Hz. Both f and ω are referred to as frequency, but ω is more precisely named *angular frequency*, measured in rad/s.

In vibrating mechanical systems modeled by (1), $u(t)$ very often represents a position or a displacement of a particular point in the system. The derivative $u'(t)$ then has the interpretation of velocity, and $u''(t)$ is the associated acceleration. The model (1) is not only applicable to vibrating mechanical systems, but also to oscillations in electrical circuits.

1.2 A centered finite difference scheme

To formulate a finite difference method for the model problem (1) we follow the four steps explained in Section 1.1.2 in [1].

Step 1: Discretizing the domain. The domain is discretized by introducing a uniformly partitioned time mesh. The points in the mesh are $t_n = n\Delta t$, $n = 0, 1, \ldots, N_t$, where $\Delta t = T/N_t$ is the constant length of the time steps. We introduce a mesh function u^n for $n = 0, 1, \ldots, N_t$, which approximates the exact solution at the mesh points. The mesh function will be computed from algebraic equations derived from the differential equation problem.

Step 2: Fulfilling the equation at discrete time points. The ODE is to be satisfied at each mesh point:

$$u''(t_n) + \omega^2 u(t_n) = 0, \quad n = 1, \ldots, N_t\,. \tag{3}$$

Step 3: Replacing derivatives by finite differences. The derivative $u''(t_n)$ is to be replaced by a finite difference approximation. A common second-order accurate approximation to the second-order derivative is

$$u''(t_n) \approx \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2}\,. \tag{4}$$

Inserting (4) in (3) yields

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} = -\omega^2 u^n\,. \tag{5}$$

We also need to replace the derivative in the initial condition by a finite difference. Here we choose a centered difference, whose accuracy is similar to the centered difference we used for u'':

$$\frac{u^1 - u^{-1}}{2\Delta t} = 0\,. \tag{6}$$

Step 4: Formulating a recursive algorithm. To formulate the computational algorithm, we assume that we have already computed u^{n-1} and u^n such that u^{n+1} is the unknown value, which we can readily solve for:

$$u^{n+1} = 2u^n - u^{n-1} - \Delta t^2\omega^2 u^n\,. \tag{7}$$

The computational algorithm is simply to apply (7) successively for $n = 1, 2, \ldots, N_t - 1$. This numerical scheme sometimes goes under the name Störmer's method or Verlet integration[1].

Computing the first step. We observe that (7) cannot be used for $n = 0$ since the computation of u^1 then involves the undefined value u^{-1} at $t = -\Delta t$. The discretization of the initial condition then comes to our rescue: (6) implies $u^{-1} = u^1$ and this relation can be combined with (7) for $n = 1$ to yield a value for u^1:

$$u^1 = 2u^0 - u^1 - \Delta t^2\omega^2 u^0,$$

which reduces to

$$u^1 = u^0 - \frac{1}{2}\Delta t^2\omega^2 u^0\,. \tag{8}$$

Exercise 5 asks you to perform an alternative derivation and also to generalize the initial condition to $u'(0) = V \neq 0$.

The computational algorithm. The steps for solving (1) becomes

1. $u^0 = I$
2. compute u^1 from (8)
3. for $n = 1, 2, \ldots, N_t - 1$:
   (a) compute u^{n+1} from (7)

The algorithm is more precisely expressed directly in Python:

```
t = linspace(0, T, Nt+1)  # mesh points in time
dt = t[1] - t[0]          # constant time step
u = zeros(Nt+1)           # solution

u[0] = I
u[1] = u[0] - 0.5*dt**2*w**2*u[0]
for n in range(1, Nt):
    u[n+1] = 2*u[n] - u[n-1] - dt**2*w**2*u[n]
```

Remark on using `w` for ω.

[1] http://en.wikipedia.org/wiki/Verlet_integration

In the code, we use `w` as the symbol for ω. The reason is that this author prefers `w` for readability and comparison with the mathematical ω instead of the full word `omega` as variable name.

Operator notation. We may write the scheme using a compact difference notation (see also Section 1.1.8 in [1]). The difference (4) has the operator notation $[D_tD_tu]^n$ such that we can write:

$$[D_tD_tu + \omega^2 u = 0]^n \,. \tag{9}$$

Note that $[D_tD_tu]^n$ means applying a central difference with step $\Delta t/2$ twice:

$$[D_t(D_tu)]^n = \frac{[D_tu]^{n+\frac{1}{2}} - [D_tu]^{n-\frac{1}{2}}}{\Delta t}$$

which is written out as

$$\frac{1}{\Delta t}\left(\frac{u^{n+1}-u^n}{\Delta t} - \frac{u^n - u^{n-1}}{\Delta t}\right) = \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2}\,.$$

The discretization of initial conditions can in the operator notation be expressed as

$$[u = I]^0, \quad [D_{2t}u = 0]^0, \tag{10}$$

where the operator $[D_{2t}u]^n$ is defined as

$$[D_{2t}u]^n = \frac{u^{n+1} - u^{n-1}}{2\Delta t}\,. \tag{11}$$

2 Implementation

2.1 Making a solver function

The algorithm from the previous section is readily translated to a complete Python function for computing and returning $u^0, u^1, \ldots, u^{N_t}$ and $t_0, t_1, \ldots, t_{N_t}$, given the input I, ω, Δt, and T:

```
import numpy as np
import matplotlib.pyplot as plt

def solver(I, w, dt, T):
    """
    Solve u'' + w**2*u = 0 for t in (0,T], u(0)=I and u'(0)=0,
    by a central finite difference method with time step dt.
    """
    dt = float(dt)
    Nt = int(round(T/dt))
    u = np.zeros(Nt+1)
    t = np.linspace(0, Nt*dt, Nt+1)

    u[0] = I
    u[1] = u[0] - 0.5*dt**2*w**2*u[0]
    for n in range(1, Nt):
        u[n+1] = 2*u[n] - u[n-1] - dt**2*w**2*u[n]
    return u, t
```


We do a simple `from module import *` to make the code as close as possible to MATLAB, although good programming habits would prefix the `numpy` and `matplotlib` calls by (abbreviations of) the module name.

A function for plotting the numerical and the exact solution is also convenient to have:

```
def u_exact(t, I, w):
    return I*np.cos(w*t)

def visualize(u, t, I, w):
    plt.plot(t, u, 'r--o')
    t_fine = np.linspace(0, t[-1], 1001)  # very fine mesh for u_e
    u_e = u_exact(t_fine, I, w)
    plt.hold('on')
    plt.plot(t_fine, u_e, 'b-')
    plt.legend(['numerical', 'exact'], loc='upper left')
    plt.xlabel('t')
    plt.ylabel('u')
    dt = t[1] - t[0]
    plt.title('dt=%g' % dt)
    umin = 1.2*u.min();  umax = -umin
    plt.axis([t[0], t[-1], umin, umax])
    plt.savefig('tmp1.png');  plt.savefig('tmp1.pdf')
```

A corresponding main program calling these functions for a simulation of a given number of periods (`num_periods`) may take the form

```
I = 1
w = 2*pi
dt = 0.05
num_periods = 5
P = 2*pi/w    #  one period
T = P*num_periods
u, t = solver(I, w, dt, T)
visualize(u, t, I, w, dt)
```

Adjusting some of the input parameters via the command line can be handy. Here is a code segment using the `ArgumentParser` tool in the `argparse` module to define option value (`-option value`) pairs on the command line:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--I', type=float, default=1.0)
parser.add_argument('--w', type=float, default=2*pi)
parser.add_argument('--dt', type=float, default=0.05)
parser.add_argument('--num_periods', type=int, default=5)
a = parser.parse_args()
I, w, dt, num_periods = a.I, a.w, a.dt, a.num_periods
```

Such parsing of the command line is explained in more detailed in Section 5.2 in [1].

A typical execution goes like

Terminal

```
Terminal> python vib_undamped.py --num_periods 20 --dt 0.1
```

Computing u'. In mechanical vibration applications one is often interested in computing the velocity $v(t) = u'(t)$ after $u(t)$ has been computed. This can be done by a central difference,

$$v(t_n) = u'(t_n) \approx v^n = \frac{u^{n+1} - u^{n-1}}{2\Delta t} = [D_{2t}u]^n \,. \tag{12}$$

This formula applies for all inner mesh points, $n = 1, \ldots, N_t - 1$. For $n = 0$, $v(0)$ is given by the initial condition on $u'(0)$, and for $n = N_t$ we can use a one-sided, backward difference:

$$v^n = [D_t^- u]^n = \frac{u^n - u^{n-1}}{\Delta t}.$$

Typical (scalar) code is

```
v = np.zeros_like(u)  # or v = np.zeros(len(u))
# Use central difference for internal points
for i in range(1, len(u)-1):
    v[i] = (u[i+1] - u[i-1])/(2*dt)
# Use initial condition for u'(0) when i=0
v[0] = 0
# Use backward difference at the final mesh point
v[-1] = (u[-1] - u[-2])/dt
```

We can get rid of the loop, which is slow for large N_t, by vectorizing the central difference. The above code segment goes as follows in its vectorized version:

```
v = np.zeros_like(u)
v[1:-1] = (u[2:] - u[:-2])/(2*dt)  # central difference
v[0] = 0                           # boundary condition u'(0)
v[-1] = (u[-1] - u[-2])/dt         # backward difference
```

2.2 Verification

Manual calculation. The simplest type of verification, which is also instructive for understanding the algorithm, is to compute u^1, u^2, and u^3 with the aid of a calculator and make a function for comparing these results with those from the `solver` function. The `test_three_steps` function in the file `vib_undamped.py`[2] shows the details how we use the hand calculations to test the code:

```
def test_three_steps():
    from math import pi
    I = 1;  w = 2*pi;  dt = 0.1;  T = 1
    u_by_hand = np.array([1.000000000000000,
                          0.802607911978213,
                          0.288358920740053])
    u, t = solver(I, w, dt, T)
    diff = np.abs(u_by_hand - u[:3]).max()
    tol = 1E-14
    assert diff < tol
```

Testing very simple solutions. Constructing test problems where the exact solution is constant or linear helps initial debugging and verification as one expects any reasonable numerical method to reproduce such solutions to machine precision. Second-order accurate methods will often also reproduce a quadratic solution. Here $[D_t D_t t^2]^n = 2$, which is the exact result. A solution $u = t^2$ leads to $u'' + \omega^2 u = 2 + (\omega t)^2 \neq 0$. We must therefore add a source in the equation: $u'' + \omega^2 u = f$ to allow a solution $u = t^2$ for $f = (\omega t)^2$. By simple insertion we can show that the mesh function $u^n = t_n^2$ is also a solution of the discrete equations. Problem 1 asks you to carry out all details to show that linear and quadratic solutions are solutions of the discrete equations. Such results are very useful for debugging and verification. You are strongly encouraged to do this problem now!

[2] `http://tinyurl.com/nm5587k/vib/vib_undamped.py`

Checking convergence rates. Empirical computation of convergence rates yields a good method for verification. The method and its computational are explained in detail in Section 3.1.6 in [1]. Readers not familiar with the concept should look up this reference before proceeding.

In the present problem, computing convergence rates means that we must

- perform m simulations with halved time steps: $\Delta t_i = 2^{-i}\Delta t_0$, $i = 0, \ldots, m-1$,
- compute the L^2 norm of the error, $E_i = \sqrt{\Delta t_i \sum_{n=0}^{N_i-1}(u^n - u_{\mathrm{e}}(t_n))^2}$ in each case,
- estimate the convergence rates r_i based on two consecutive experiments $(\Delta t_{i-1}, E_{i-1})$ and $(\Delta t_i, E_i)$, assuming $E_i = C(\Delta t_i)^r$ and $E_{i-1} = C(\Delta t_{i-1})^r$. From these equations it follows that $r = \ln(E_{i-1}/E_i)/\ln(\Delta t_{i-1}/\Delta t_i)$. Since this r will vary with i, we equip it with an index and call it r_{i-1}, where i runs from 1 to $m-1$.

The computed rates $r_0, r_1, \ldots, r_{m-2}$ hopefully converges to a number, which hopefully is 2, the right one, in the present problem. The convergence of the rates demands that the time steps Δt_i are sufficiently small for the error model $E_i = (\Delta t_i)^r$ to be valid.

All the implementational details of computing the sequence $r_0, r_1, \ldots, r_{m-2}$ appear below.

```
def convergence_rates(m, solver_function, num_periods=8):
    """
    Return m-1 empirical estimates of the convergence rate
    based on m simulations, where the time step is halved
    for each simulation.
    solver_function(I, w, dt, T) solves each problem, where T
    is based on simulation for num_periods periods.
    """
    from math import pi
    w = 0.35; I = 0.3       # just chosen values
    P = 2*pi/w              # period
    dt = P/30               # 30 time step per period 2*pi/w
    T = P*num_periods

    dt_values = []
    E_values = []
    for i in range(m):
        u, t = solver_function(I, w, dt, T)
        u_e = u_exact(t, I, w)
        E = np.sqrt(dt*np.sum((u_e-u)**2))
        dt_values.append(dt)
        E_values.append(E)
        dt = dt/2

    r = [np.log(E_values[i-1]/E_values[i])/
         np.log(dt_values[i-1]/dt_values[i])
         for i in range(1, m, 1)]
    return r
```

The expected convergence rate is 2, because we have used a second-order finite difference approximations $[D_t D_t u]^n$ to the ODE and a second-order finite difference formula for the initial condition for u'. Other theoretical error measures also points to $r = 2$.

In the present problem, when Δt_0 corresponds to 30 time steps per period, the returned `r` list has all its values equal to 2.00 (if rounded to two decimals). This amazing result means that all Δt_i values are well into the asymptotic regime where the error model $E_i = C(\Delta t_i)^r$ is valid.

We can now construct a test function that computes convergence rates and checks that the final (and usually the best) estimate is sufficiently close to 2. Here, a rough tolerance of 0.1 is enough. This unit test goes like

```
def test_convergence_rates():
    r = convergence_rates(m=5, solver_function=solver, num_periods=8)
    # Accept rate to 1 decimal place
    tol = 0.1
    assert abs(r[-1] - 2.0) < tol
```

The complete code appears in the file `vib_undamped.py`.

2.3 Scaled model

It is advantageous to use dimensionless variables in simulations, because fewer parameters need to be set. The present problem is made dimensionless by introducing dimensionless variables $\bar{t} = t/t_c$ and $\bar{u} = u/u_c$, where t_c and u_c are characteristic scales for t and u, respectively. The scaled ODE problem reads

$$\frac{u_c}{t_c^2}\frac{d^2\bar{u}}{d\bar{t}^2} + u_c\bar{u} = 0, \quad u_c\bar{u}(0) = I,\ \frac{u_c}{t_c}\frac{d\bar{u}}{d\bar{t}}(0) = 0\,.$$

A common choice is to take t_c as one period of the oscillations, $t_c = 2\pi/w$, and $u_c = I$. This gives the dimensionless model

$$\frac{d^2\bar{u}}{\bar{t}^2} + 4\pi^2\bar{u} = 0, \quad \bar{u}(0) = 1,\ \bar{u}'(0) = 0\,. \tag{13}$$

Observe that there are no physical parameters in (13)! We can therefore perform a single numerical simulation $\bar{u}(\bar{t})$ and afterwards recover any $u(t;\omega,I)$ by

$$u(t;\omega,I) = u_c\bar{u}(t/t_c) = I\bar{u}(omegat/(2\pi))\,.$$

We can easily check this assertion: the solution of the scaled problem is $\bar{u}(\bar{t}) = \cos(2\pi\bar{t})$. The formula for u in terms of $\bar{u}$ gives $u = I\cos(\omega t)$, which is nothing but the solution of the original problem with dimensions.

The scaled model can by run by calling `solver(I=1, w=2*pi, dt, T)`. Each period is now 1 and `T` simply counts the number of periods. Choosing `dt` as `1./M` gives `M` time steps per period.

3 Long time simulations

Figure 1 shows a comparison of the exact and numerical solution for the scaled model (13) with $\Delta t = 0.1, 0.05$. From the plot we make the following observations:

- The numerical solution seems to have correct amplitude.
- There is a angular frequency error which is reduced by reducing the time step.
- The total angular frequency error grows with time.

By angular frequency error we mean that the numerical angular frequency differs from the exact ω. This is evident by looking at the peaks of the numerical solution: these have incorrect positions compared with the peaks of the exact cosine solution. The effect can be mathematical expressed by writing the numerical solution as $I\cos\tilde{\omega}t$, where $\tilde{\omega}$ is not exactly equal to ω. Later, we shall mathematically quantify this numerical angular frequency $\tilde{\omega}$.

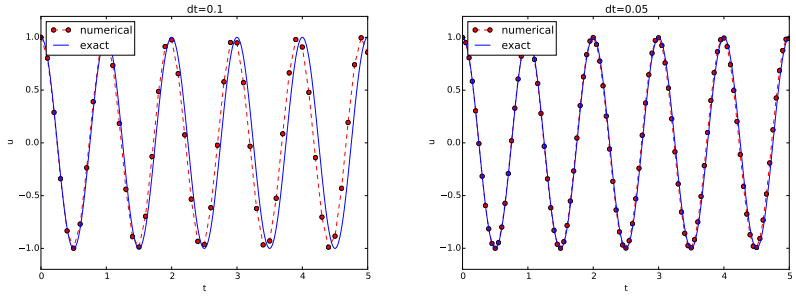


Figure 1: Effect of halving the time step.

3.1 Using a moving plot window

In vibration problems it is often of interest to investigate the system's behavior over long time intervals. Errors in the angular frequency accumulate and become more visible as time grows. We can investigate long time series by introducing a moving plot window that can move along with the p most recently computed periods of the solution. The SciTools[3] package contains a convenient tool for this: `MovingPlotWindow`. Typing `pydoc scitools.MovingPlotWindow` shows a demo and a description of its use. The function below utilizes the moving plot window and is in fact called by the `main` function the `vib_undamped` module if the number of periods in the simulation exceeds 10.

```
def visualize_front(u, t, I, w, savefig=False, skip_frames=1):
    """
    Visualize u and the exact solution vs t, using a
    moving plot window and continuous drawing of the
    curves as they evolve in time.
    Makes it easy to plot very long time series.
    Plots are saved to files if savefig is True.
    Only each skip_frames-th plot is saved (e.g., if
    skip_frame=10, only each 10th plot is saved to file;
    this is convenient if plot files corresponding to
    different time steps are to be compared).
    """
    import scitools.std as st
    from scitools.MovingPlotWindow import MovingPlotWindow
    from math import pi

    # Remove all old plot files tmp_*.png
    import glob, os
    for filename in glob.glob('tmp_*.png'):
        os.remove(filename)

    P = 2*pi/w  # one period
    umin = 1.2*u.min();  umax = -umin
    dt = t[1] - t[0]
    plot_manager = MovingPlotWindow(
        window_width=8*P,
        dt=dt,
        yaxis=[umin, umax],
        mode='continuous drawing')
```

[3] https://github.com/hplgit/scitools

```
    frame_counter = 0
    for n in range(1,len(u)):
        if plot_manager.plot(n):
            s = plot_manager.first_index_in_plot
            st.plot(t[s:n+1], u[s:n+1], 'r-1',
                    t[s:n+1], I*cos(w*t)[s:n+1], 'b-1',
                    title='t=%6.3f' % t[n],
                    axis=plot_manager.axis(),
                    show=not savefig) # drop window if savefig
            if savefig and n % skip_frames == 0:
                filename = 'tmp_%04d.png' % frame_counter
                st.savefig(filename)
                print 'making plot file', filename, 'at t=%g' % t[n]
                frame_counter += 1
        plot_manager.update(n)
```

We run the scaled problem (the default values for the command-line arguments `-I` and `-w` correspond to the scaled problem) for 40 periods with 20 time steps per period:

Terminal

```
Terminal> python vib_undamped.py --dt 0.05 --num_periods 40
```

The moving plot window is invoked, and we can follow the numerical and exact solutions as time progresses. From this demo we see that the angular frequency error is small in the beginning, but it becomes more prominent with time. A new run with $\Delta t = 0.1$ (i.e., only 10 time steps per period) clearly shows that the phase errors become significant even earlier in the time series, deteriorating the solution further.

3.2 Making animations

Producing standard video formats. The `visualize_front` function stores all the plots in files whose names are numbered: `tmp_0000.png`, `tmp_0001.png`, `tmp_0002.png`, and so on. From these files we may make a movie. The Flash format is popular,

Terminal

```
Terminal> ffmpeg -r 12 -i tmp_%04d.png -c:v flv movie.flv
```

The `ffmpeg` program can be replaced by the `avconv` program in the above command if desired (but at the time of this writing it seems to be more momentum in the `ffmpeg` project). The `-r` option should come first and describes the number of frames per second in the movie. The `-i` option describes the name of the plot files. Other formats can be generated by changing the video codec and equipping the video file with the right extension:

Format	Codec and filename
Flash	`-c:v flv movie.flv`
MP4	`-c:v libx264 movie.mp4`
WebM	`-c:v libvpx movie.webm`
Ogg	`-c:v libtheora movie.ogg`

The video file can be played by some video player like `vlc`, `mplayer`, `gxine`, or `totem`, e.g.,

Terminal

```
Terminal> vlc movie.webm
```


A web page can also be used to play the movie. Today's standard is to use the HTML5 `video` tag:

```
<video autoplay loop controls
       width='640' height='365' preload='none'>
<source src='movie.webm'  type='video/webm; codecs="vp8, vorbis"'>
</video>
```

Modern browsers do not support all of the video formats. MP4 is needed to successfully play the videos on Apple devices that use the Safari browser. WebM is the preferred format for Chrome, Opera, Firefox, and Internet Explorer v9+. Flash was a popular format, but older browsers that required Flash can play MP4. All browsers that work with Ogg can also work with WebM. This means that to have a video work in all browsers, the video should be available in the MP4 and WebM formats. The proper HTML code reads

```
<video autoplay loop controls
       width='640' height='365' preload='none'>
<source src='movie.mp4'   type='video/mp4;
 codecs="avc1.42E01E, mp4a.40.2"'>
<source src='movie.webm'  type='video/webm;
 codecs="vp8, vorbis"'>
</video>
```

The MP4 format should appear first to ensure that Apple devices will load the video correctly.

Caution: number the plot files correctly.

To ensure that the individual plot frames are shown in correct order, it is important to number the files with zero-padded numbers (0000, 0001, 0002, etc.). The printf format `%04d` specifies an integer in a field of width 4, padded with zeros from the left. A simple Unix wildcard file specification like `tmp_*.png` will then list the frames in the right order. If the numbers in the filenames were not zero-padded, the frame `tmp_11.png` would appear before `tmp_2.png` in the movie.

Paying PNG files in a web browser. The `scitools movie` command can create a movie player for a set of PNG files such that a web browser can be used to watch the movie. This interface has the advantage that the speed of the movie can easily be controlled, a feature that scientists often appreciate. The command for creating an HTML with a player for a set of PNG files `tmp_*.png` goes like

Terminal

```
Terminal> scitools movie output_file=vib.html fps=4 tmp_*.png
```

The `fps` argument controls the speed of the movie ("frames per second").

To watch the movie, load the video file `vib.html` into some browser, e.g.,

Terminal

```
Terminal> google-chrome vib.html  # invoke web page
```

Clicking on `Start movie` to see the result. Moving this movie to some other place requires moving `vib.html` *and all the PNG files* `tmp_*.png`:

```
Terminal> mkdir vib_dt0.1
Terminal> mv tmp_*.png vib_dt0.1
Terminal> mv vib.html vib_dt0.1/index.html
```

Making animated GIF files. The `convert` program from the ImageMagick software suite can be used to produce animated GIF files from a set of PNG files:

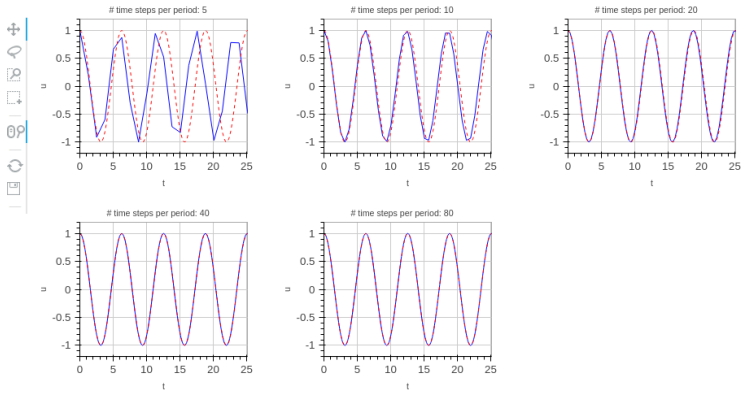
```
Terminal> convert -delay 25 tmp_vib*.png tmp_vib.gif
```

The `-delay` option needs an argument of the delay between each frame, measured in 1/100 s, so 4 frames/s here gives 25/100 s delay. Note, however, that in this particular example with $\Delta t = 0.05$ and 40 periods, making an animated GIF file out of the large number of PNG files is a very heavy process and not considered feasible. Animated GIFs are best suited for animations with not so many frames and where you want to see each frame and play them slowly.

3.3 Using Bokeh to compare graphs

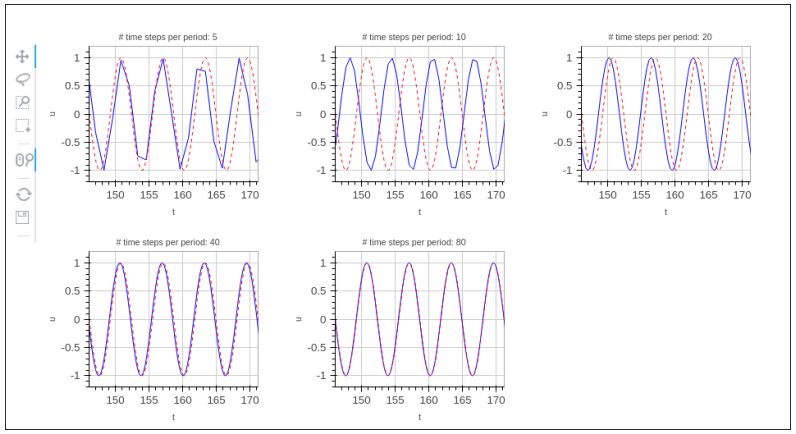
Instead of a moving plot frame, one can use tools that allows panning by the mouse. For example, we can show four periods of a signal in a plot and then scroll with the mouse through the rest of the simulation. The Bokeh[4] plotting library offers such tools, but the plot must be displayed in a web browser. The documentation of Bokeh is excellent, so here we just show how the library can be used to compare a set of u curves corresponding to long time simulations.

Imagine we have performed experiments for a set of Δt values. We want each curve, together with the exact solution, to appear in a plot, and then arrange all plots in a grid-like fashion:



Furthermore, we want the axis to couple such that if we move into the future in one plot, all the other plots follows (note the displaced t axes!):

[4] `http://bokeh.pydata.org/en/latest/docs/quickstart.html`



A function for creating a Bokeh plot, given a list of `u` arrays and corresponding `t` arrays, from different simulations, described compactly in a list of strings `legends`, takes the following form:

```
def bokeh_plot(u, t, legends, I, w, t_range, filename):
    """
    Make plots for u vs t using the Bokeh library.
    u and t are lists (several experiments can be compared).
    legens contain legend strings for the various u,t pairs.
    """
    if not isinstance(u, (list,tuple)):
        u = [u]  # wrap in list
    if not isinstance(t, (list,tuple)):
        t = [t]  # wrap in list
    if not isinstance(legends, (list,tuple)):
        legends = [legends]  # wrap in list

    import bokeh.plotting as plt
    plt.output_file(filename, mode='cdn', title='Comparison')
    # Assume that all t arrays have the same range
    t_fine = np.linspace(0, t[0][-1], 1001)  # fine mesh for u_e
    tools = 'pan,wheel_zoom,box_zoom,reset,'\
            'save,box_select,lasso_select'
    u_range = [-1.2*I, 1.2*I]
    font_size = '8pt'
    p = []  # list of plot objects
    # Make the first figure
    p_ = plt.figure(
        width=300, plot_height=250, title=legends[0],
        x_axis_label='t', y_axis_label='u',
        x_range=t_range, y_range=u_range, tools=tools,
        title_text_font_size=font_size)
    p_.xaxis.axis_label_text_font_size=font_size
    p_.yaxis.axis_label_text_font_size=font_size
    p_.line(t[0], u[0], line_color='blue')
    # Add exact solution
    u_e = u_exact(t_fine, I, w)
    p_.line(t_fine, u_e, line_color='red', line_dash='4 4')
    p.append(p_)
    # Make the rest of the figures and attach their axes to
    # the first figure's axes
    for i in range(1, len(t)):
```

```
        p_ = plt.figure(
            width=300, plot_height=250, title=legends[i],
            x_axis_label='t', y_axis_label='u',
            x_range=p[0].x_range, y_range=p[0].y_range, tools=tools,
            title_text_font_size=font_size)
        p_.xaxis.axis_label_text_font_size = font_size
        p_.yaxis.axis_label_text_font_size = font_size
        p_.line(t[i], u[i], line_color='blue')
        p_.line(t_fine, u_e, line_color='red', line_dash='4 4')
        p.append(p_)

    # Arrange all plots in a grid with 3 plots per row
    grid = [[]]
    for i, p_ in enumerate(p):
        grid[-1].append(p_)
        if (i+1) % 3 == 0:
            # New row
            grid.append([])
    plot = plt.gridplot(grid, toolbar_location='left')
    plt.save(plot)
    plt.show(plot)
```

A particular example using the `bokeh_plot` function appears below.

```
def demo_bokeh():
    """Solve a scaled ODE u'' + u = 0."""
    from math import pi
    w = 1.0        # Scaled problem (frequency)
    P = 2*np.pi/w  # Period
    num_steps_per_period = [5, 10, 20, 40, 80]
    T = 40*P       # Simulation time: 40 periods
    u = []         # List of numerical solutions
    t = []         # List of corresponding meshes
    legends = []
    for n in num_steps_per_period:
        dt = P/n
        u_, t_ = solver(I=1, w=w, dt=dt, T=T)
        u.append(u_)
        t.append(t_)
        legends.append('# time steps per period: %d' % n)
    bokeh_plot(u, t, legends, I=1, w=w, t_range=[0, 4*P],
               filename='tmp.html')
```

3.4 Using a line-by-line ascii plotter

Plotting functions vertically, line by line, in the terminal window using ascii characters only is a simple, fast, and convenient visualization technique for long time series. Note that the time axis then is positive downwards on the screen. The tool `scitools.avplotter.Plotter` makes it easy to create such plots:

```
def visualize_front_ascii(u, t, I, w, fps=10):
    """
    Plot u and the exact solution vs t line by line in a
    terminal window (only using ascii characters).
    Makes it easy to plot very long time series.
    """
    from scitools.avplotter import Plotter
    import time
    from math import pi
    P = 2*pi/w
    umin = 1.2*u.min();  umax = -umin

    p = Plotter(ymin=umin, ymax=umax, width=60, symbols='+o')
```



```
    for n in range(len(u)):
        print p.plot(t[n], u[n], I*cos(w*t[n])), \
              '%.1f' % (t[n]/P)
        time.sleep(1/float(fps))
```

The call `p.plot` returns a line of text, with the t axis marked and a symbol `+` for the first function (`u`) and `o` for the second function (the exact solution). Here we append to this text a time counter reflecting how many periods the current time point corresponds to. A typical output ($\omega = 2\pi$, $\Delta t = 0.05$) looks like this:

```
                              |                        o+      14.0
                              |                      + o       14.0
                              |                   +    o       14.1
                              |              +     o           14.1
                              |     +         o                14.2
                             +|       o                       14.2
                        +     |                                14.2
                   +       o  |                                14.3
              +     o         |                                14.4
         +   o                |                                14.4
        +o                    |                                14.5
        o +                   |                                14.5
         o    +               |                                14.6
             o     +          |                                14.6
                  o       +   |                                14.7
                        o     | +                              14.7
                              |       +                        14.8
                              |       o       +                14.8
                              |               o     +          14.9
                              |                     o   +      14.9
                              |                         o+     15.0
```

3.5 Empirical analysis of the solution

For oscillating functions like those in Figure 1 we may compute the amplitude and frequency (or period) empirically. That is, we run through the discrete solution points (t_n, u_n) and find all maxima and minima points. The distance between two consecutive maxima (or minima) points can be used as estimate of the local period, while half the difference between the u value at a maximum and a nearby minimum gives an estimate of the local amplitude.

The local maxima are the points where

$$u^{n-1} < u^n > u^{n+1}, \quad n = 1, \ldots, N_t - 1, \tag{14}$$

and the local minima are recognized by

$$u^{n-1} > u^n < u^{n+1}, \quad n = 1, \ldots, N_t - 1\,. \tag{15}$$

In computer code this becomes

```
def minmax(t, u):
    minima = []; maxima = []
    for n in range(1, len(u)-1, 1):
        if u[n-1] > u[n] < u[n+1]:
            minima.append((t[n], u[n]))
        if u[n-1] < u[n] > u[n+1]:
            maxima.append((t[n], u[n]))
    return minima, maxima
```

Note that the two returned objects are lists of tuples.

Let (t_i, e_i), $i = 0, \ldots, M-1$, be the sequence of all the M maxima points, where t_i is the time value and e_i the corresponding u value. The local period can be defined as $p_i = t_{i+1} - t_i$. With Python syntax this reads

```
def periods(maxima):
    p = [extrema[n][0] - maxima[n-1][0]
         for n in range(1, len(maxima))]
    return np.array(p)
```

The list `p` created by a list comprehension is converted to an array since we probably want to compute with it, e.g., find the corresponding frequencies `2*pi/p`.

Having the minima and the maxima, the local amplitude can be calculated as the difference between two neighboring minimum and maximum points:

```
def amplitudes(minima, maxima):
    a = [(abs(maxima[n][1] - minima[n][1]))/2.0
         for n in range(min(len(minima),len(maxima)))]
    return np.array(a)
```

The code segments are found in the file `vib_empirical_analysis.py`[5].

Since `a[i]` and `p[i]` correspond to the i-th amplitude estimate and the i-th period estimate, respectively, it is most convenient to visualize the `a` and `p` values with the index `i` on the horizontal axis. (There is no unique time point associated with either of these estimate since values at two different time points were used in the computations.)

In the analysis of very long time series, it is advantageous to compute and plot `p` and `a` instead of u to get an impression of the development of the oscillations. Let us do this for the scaled problem and $\Delta t = 0.1, 0.05, 0.01$. A ready-made function

```
plot_empirical_freq_and_amplitude(u, t, I, w)
```

computes the empirical amplitudes and periods, and creates a plot where the amplitudes and angular frequencies are visualized together with the exact amplitude `I` and the exact angular frequency `w`. We can make a little program for creating the plot:

```
from vib_undamped import solver, plot_empirical_freq_and_amplitude
from math import pi
dt_values = [0.1, 0.05, 0.01]
u_cases = []
t_cases = []
for dt in dt_values:
    # Simulate scaled problem for 40 periods
    u, t = solver(I=1, w=2*pi, dt=dt, T=40)
    u_cases.append(u)
    t_cases.append(t)
plot_empirical_freq_and_amplitude(u_cases, t_cases, I=1, w=2*pi)
```

Figure 2 shows the result: we clearly see that lowering Δt improves the angular frequency significantly, while the amplitude seems to be more accurate. The lines with $\Delta t = 0.01$, corresponding to 100 steps per period, can hardly be distinguished from the exact values. The next section shows how we can get mathematical insight into why amplitudes are good and frequencies are more inaccurate.

[5] `http://tinyurl.com/nm5587k/vib/vib_empirical_analysis.py`

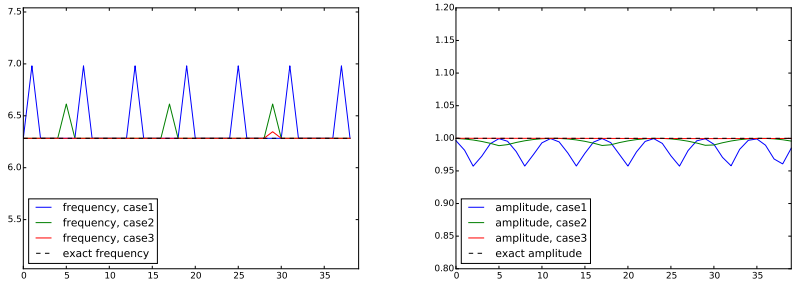


Figure 2: Empirical amplitude and angular frequency for three cases of time steps.

4 Analysis of the numerical scheme

4.1 Deriving a solution of the numerical scheme

After having seen the phase error grow with time in the previous section, we shall now quantify this error through mathematical analysis. The key tool in the analysis will be to establish an exact solution of the discrete equations. The difference equation (7) has constant coefficients and is homogeneous. Such equations are known to have solutions on the form $u^n = CA^n$, where A is some number to be determined from the difference equation and C is found as the initial condition ($C = I$). Recall that n in u^n is a superscript labeling the time level, while n in A^n is an exponent.

With oscillating functions as solutions, the algebra will be considerably simplified if we seek an A on the form

$$A = e^{i\tilde{\omega}\Delta t},$$

and solve for the numerical frequency $\tilde{\omega}$ rather than A. Note that $i = \sqrt{-1}$ is the imaginary unit. (Using a complex exponential function gives simpler arithmetics than working with a sine or cosine function.) We have

$$A^n = e^{i\tilde{\omega}\Delta t\, n} = e^{i\tilde{\omega}t} = \cos(\tilde{\omega}t) + i\sin(\tilde{\omega}t)\,.$$

The physically relevant numerical solution can be taken as the real part of this complex expression.

The calculations go as

$$
\begin{aligned}
[D_tD_tu]^n &= \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} \\
&= I\frac{A^{n+1} - 2A^n + A^{n-1}}{\Delta t^2} \\
&= \frac{I}{\Delta t^2}(e^{i\tilde{\omega}(t+\Delta t)} - 2e^{i\tilde{\omega}t} + e^{i\tilde{\omega}(t-\Delta t)}) \\
&= Ie^{i\tilde{\omega}t}\frac{1}{\Delta t^2}\left(e^{i\tilde{\omega}\Delta t} + e^{i\tilde{\omega}(-\Delta t)} - 2\right) \\
&= Ie^{i\tilde{\omega}t}\frac{2}{\Delta t^2}\left(\cosh(i\tilde{\omega}\Delta t) - 1\right) \\
&= Ie^{i\tilde{\omega}t}\frac{2}{\Delta t^2}\left(\cos(\tilde{\omega}\Delta t) - 1\right) \\
&= -Ie^{i\tilde{\omega}t}\frac{4}{\Delta t^2}\sin^2(\frac{\tilde{\omega}\Delta t}{2})
\end{aligned}
$$

The last line follows from the relation $\cos x - 1 = -2\sin^2(x/2)$ (try `cos(x)-1` in wolframalpha.com[6] to see the formula).

The scheme (7) with $u^n = Ie^{i\omega\tilde{\Delta}t\,n}$ inserted now gives

$$
-Ie^{i\tilde{\omega}t}\frac{4}{\Delta t^2}\sin^2(\frac{\tilde{\omega}\Delta t}{2}) + \omega^2 Ie^{i\tilde{\omega}t} = 0, \tag{16}
$$

which after dividing by $Ie^{i\tilde{\omega}t}$ results in

$$
\frac{4}{\Delta t^2}\sin^2(\frac{\tilde{\omega}\Delta t}{2}) = \omega^2\,. \tag{17}
$$

The first step in solving for the unknown $\tilde{\omega}$ is

$$
\sin^2(\frac{\tilde{\omega}\Delta t}{2}) = \left(\frac{\omega\Delta t}{2}\right)^2\,.
$$

Then, taking the square root, applying the inverse sine function, and multiplying by $2/\Delta t$, results in

$$
\tilde{\omega} = \pm\frac{2}{\Delta t}\sin^{-1}\left(\frac{\omega\Delta t}{2}\right)\,. \tag{18}
$$

The first observation of (18) tells that there is a phase error since the numerical frequency $\tilde{\omega}$ never equals the exact frequency ω. But how good is the approximation (18)? That is, what is the error $\omega - \tilde{\omega}$ or $\tilde{\omega}/\omega$? Taylor series expansion for small Δt may give an expression that is easier to understand than the complicated function in (18):

```
>>> from sympy import *
>>> dt, w = symbols('dt w')
>>> w_tilde_e = 2/dt*asin(w*dt/2)
>>> w_tilde_series = w_tilde_e.series(dt, 0, 4)
>>> print w_tilde_series
w + dt**2*w**3/24 + O(dt**4)
```

This means that

[6] `http://www.wolframalpha.com`

$$
\tilde{\omega} = \omega\left(1 + \frac{1}{24}\omega^2\Delta t^2\right) + \mathcal{O}(\Delta t^4)\,. \tag{19}
$$

The error in the numerical frequency is of second-order in Δt, and the error vanishes as $\Delta t \to 0$. We see that $\tilde{\omega} > \omega$ since the term $\omega^3\Delta t^2/24 > 0$ and this is by far the biggest term in the series expansion for small $\omega\Delta t$. A numerical frequency that is too large gives an oscillating curve that oscillates too fast and therefore "lags behind" the exact oscillations, a feature that can be seen in the left plot in Figure 1.

Figure 3 plots the discrete frequency (18) and its approximation (19) for $\omega = 1$ (based on the program `vib_plot_freq.py`[7]). Although $\tilde{\omega}$ is a function of Δt in (19), it is misleading to think of Δt as the important discretization parameter. It is the product $\omega\Delta t$ that is the key discretization parameter. This quantity reflects the *number of time steps per period* of the oscillations. To see this, we set $P = N_P\Delta t$, where P is the length of a period, and N_P is the number of time steps during a period. Since P and ω are related by $P = 2\pi/\omega$, we get that $\omega\Delta t = 2\pi/N_P$, which shows that $\omega\Delta t$ is directly related to N_P.

The plot shows that at least $N_P \sim 25 - 30$ points per period are necessary for reasonable accuracy, but this depends on the length of the simulation (T) as the total phase error due to the frequency error grows linearly with time (see Exercise 2).

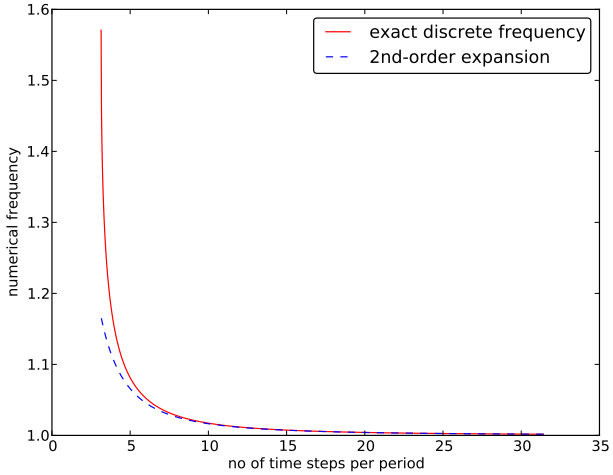


Figure 3: Exact discrete frequency and its second-order series expansion.

[7] `http://tinyurl.com/nm5587k/vib/vib_plot_freq.py`

4.2 Exact discrete solution

Perhaps more important than the $\tilde{\omega} = \omega + \mathcal{O}(\Delta t^2)$ result found above is the fact that we have an exact discrete solution of the problem:

$$u^n = I\cos\left(\tilde{\omega} n\Delta t\right), \quad \tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(\frac{\omega\Delta t}{2}\right). \tag{20}$$

We can then compute the error mesh function

$$e^n = u_{\mathrm{e}}(t_n) - u^n = I\cos\left(\omega n\Delta t\right) - I\cos\left(\tilde{\omega} n\Delta t\right). \tag{21}$$

From the formula $\cos 2x - \cos 2y = -2\sin(x-y)\sin(x+y)$ we can rewrite e^n so the expression is easier to interpret:

$$e^n = -2I\sin\left(t\frac{1}{2}\left(\omega - \tilde{\omega}\right)\right)\sin\left(t\frac{1}{2}\left(\omega + \tilde{\omega}\right)\right). \tag{22}$$

The error mesh function is ideal for verification purposes and you are strongly encouraged to make a test based on (20) by doing Exercise 10.

4.3 Convergence

We can use (19), (21), or (22) to show *convergence* of the numerical scheme, i.e., $e^n \to 0$ as $\Delta t \to 0$. We have that

$$\lim_{\Delta t\to 0}\tilde{\omega} = \lim_{\Delta t\to 0}\frac{2}{\Delta t}\sin^{-1}\left(\frac{\omega\Delta t}{2}\right) = \omega,$$

by L'Hopital's rule or simply asking `sympy` or WolframAlpha[8] about the limit:

```
>>> import sympy as sym
>>> dt, w = sym.symbols('x w')
>>> sym.limit((2/dt)*sym.asin(w*dt/2), dt, 0, dir='+')
w
```

Also (19) can be used to establish this result that $\tilde{\omega} \to \omega$. It then follows from the expression(s) for e^n that $e^n \to 0$.

4.4 The global error

To achieve more analytical insight into the nature of the global error, we can Taylor expand the error mesh function (21). Since $\tilde{\omega}$ in (18) contains Δt in the denominator we use the series expansion for $\tilde{\omega}$ inside the cosine function. A relevant `sympy` session is

```
>>> from sympy import *
>>> dt, w, t = symbols('dt w t')
>>> w_tilde_e = 2/dt*asin(w*dt/2)
>>> w_tilde_series = w_tilde_e.series(dt, 0, 4)
>>> w_tilde_series
w + dt**2*w**3/24 + O(dt**4)
```

Series expansions in `sympy` have the inconvenient `O()` term that prevents further calculations with the series. We can use the `removeO()` command to get rid of the `O()` term:

[8] `http://www.wolframalpha.com/input/?i=%282%2Fx%29*asin%28w*x%2F2%29+as+x-%3E0`


```
>>> w_tilde_series = w_tilde_series.removeO()
>>> w_tilde_series
dt**2*w**3/24 + w
```

Using this `w_tilde_series` expression for $\tilde{w}$ in (21), dropping I (which is a common factor), and performing a series expansion of the error yields

```
>>> error = cos(w*t) - cos(w_tilde_series*t)
>>> error.series(dt, 0, 6)
dt**2*t*w**3*sin(t*w)/24 + dt**4*t**2*w**6*cos(t*w)/1152 + O(dt**6)
```

Since we are mainly interested in the leading-order term in such expansions (the term with lowest power in Δt and goes most slowly to zero), we use the `.as_leading_term(dt)` construction to pick out this term:

```
>>> error.series(dt, 0, 6).as_leading_term(dt)
dt**2*t*w**3*sin(t*w)/24
```

The last result means that the leading order global (true) error at a point t is proportional to $\omega^3 t\Delta t^2$. Now, t is related to Δt through $t = n\Delta t$. The factor $\sin(\omega t)$ can at most be 1, so we use this value to bound the leading-order expression to its maximum value

$$e^n = \frac{1}{24}n\omega^3\Delta t^3.$$

This is the dominating term of the error *at a point*.

We are interested in the accumulated global error, which can be taken as the ℓ^2 norm of e^n. The norm is simply computed by summing contributions from all mesh points:

$$||e^n||_{\ell^2}^2 = \Delta t\sum_{n=0}^{N_t}\frac{1}{24^2}n^2\omega^6\Delta t^6 = \frac{1}{24^2}\omega^6\Delta t^7\sum_{n=0}^{N_t}n^2.$$

The sum $\sum_{n=0}^{N_t} n^2$ is approximately equal to $\frac{1}{3}N_t^3$. Replacing N_t by $T/\Delta t$ and taking the square root gives the expression

$$||e^n||_{\ell^2} = \frac{1}{24}\sqrt{\frac{T^3}{3}}\omega^3\Delta t^2.$$

This is our expression for the global (or integrated) error. The main result from this expression is that also the global error is proportional to Δt^2.

4.5 Stability

Looking at (20), it appears that the numerical solution has constant and correct amplitude, but an error in the angular frequency. A constant amplitude is not necessarily the case, however! To see this, note that if only Δt is large enough, the magnitude of the argument to $\sin^{-1}$ in (18) may be larger than 1, i.e., $\omega\Delta t/2 > 1$. In this case, $\sin^{-1}(\omega\Delta t/2)$ has a complex value and therefore $\tilde{\omega}$ becomes complex. Type, for example, `asin(x)` in wolframalpha.com[9] to see basic properties of $\sin^{-1}(x)$).

A complex $\tilde{\omega}$ can be written $\tilde{\omega} = \tilde{\omega}_r + i\tilde{\omega}_i$. Since $\sin^{-1}(x)$ has a *negative* imaginary part for $x > 1$, $\tilde{\omega}_i < 0$, which means that $e^{i\tilde{\omega}t} = e^{-\tilde{\omega}_i t}e^{i\tilde{\omega}_r t}$ will lead to exponential growth in time because $e^{-\tilde{\omega}_i t}$ with $\tilde{\omega}_i < 0$ has a positive exponent.

[9] `http://www.wolframalpha.com`

Stability criterion.

We do not tolerate growth in the amplitude since such growth is not present in the exact solution. Therefore, we must impose a *stability criterion* that the argument in the inverse sine function leads to real and not complex values of $\tilde{\omega}$. The stability criterion reads

$$\frac{\omega\Delta t}{2} \leq 1 \quad \Rightarrow \quad \Delta t \leq \frac{2}{\omega}. \tag{23}$$

With $\omega = 2\pi$, $\Delta t > \pi^{-1} = 0.3183098861837907$ will give growing solutions. Figure 4 displays what happens when $\Delta t = 0.3184$, which is slightly above the critical value: $\Delta t = \pi^{-1} + 9.01 \cdot 10^{-5}$.

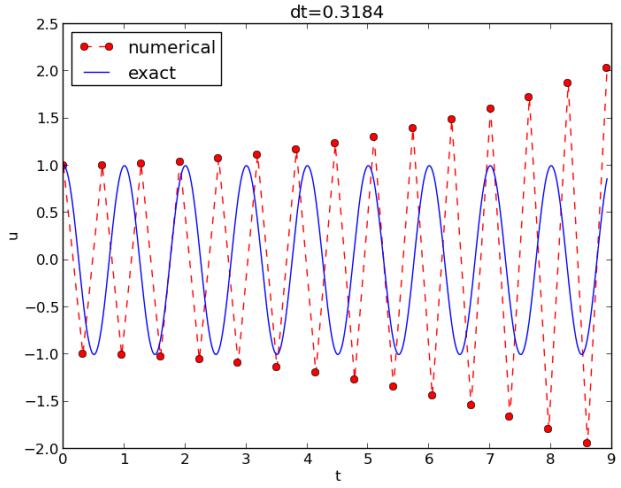


Figure 4: Growing, unstable solution because of a time step slightly beyond the stability limit.

4.6 About the accuracy at the stability limit

An interesting question is whether the stability condition $\Delta t < 2/\omega$ is unfortunate, or more precisely: would it be meaningful to take larger time steps to speed up computations? The answer is a clear no. At the stability limit, we have that $\sin^{-1}\omega\Delta t/2 = \sin^{-1} 1 = \pi/2$, and therefore $\tilde{\omega} = \pi/\Delta t$. (Note that the approximate formula (19) is very inaccurate for this value of Δt as it predicts $\tilde{\omega} = 2.34/pi$, which is a 25 percent reduction.) The corresponding period of the numerical solution is $\tilde{P} = 2\pi/\tilde{\omega} = 2\Delta t$, which means that there is just one time step Δt between a peak

(maximum) and a through[10] (minimum) in the numerical solution. This is the shortest possible wave that can be represented in the mesh! In other words, it is not meaningful to use a larger time step than the stability limit.

Also, the error in angular frequency when $\Delta t = 2/\omega$ is severe: Figure 5 shows a comparison of the numerical and analytical solution with $\omega = 2\pi$ and $\Delta t = 2/\omega = \pi^{-1}$. Already after one period, the numerical solution has a through while the exact solution has a peak (!). The error in frequency when Δt is at the stability limit becomes $\omega - \tilde{\omega} = \omega(1 - \pi/2) \approx -0.57\omega$. The corresponding error in the period is $P - \tilde{P} \approx 0.36P$. The error after m periods is then $0.36mP$. This error has reached half a period when $m = 1/(2 \cdot 0.36) \approx 1.38$, which theoretically confirms the observations in Figure 5 that the numerical solution is a through ahead of a peak already after one and a half period. Consequently, Δt should be chosen much less than the stability limit to achieve meaningful numerical computations.

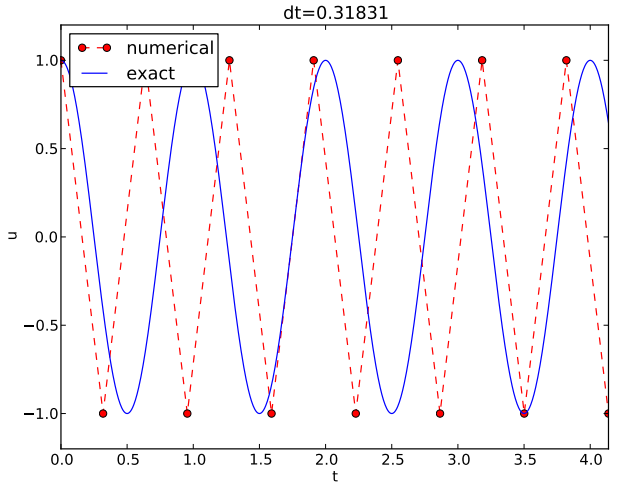


Figure 5: Numerical solution with Δt exactly at the stability limit.

Summary.

From the accuracy and stability analysis we can draw three important conclusions:

1. The key parameter in the formulas is $p = \omega\Delta t$. The period of oscillations is $P = 2\pi/\omega$, and the number of time steps per period is $N_P = P/\Delta t$. Therefore, $p = \omega\Delta t = 2\pi N_P$,

[10] https://simple.wikipedia.org/wiki/Wave_(physics)

showing that the critical parameter is the number of time steps per period. The smallest possible N_P is 2, showing that $p \in (0, \pi]$.

2. Provided $p \leq 2$, the amplitude of the numerical solution is constant.

3. The ratio of the numerical angular frequency and the exact one is $\tilde{\omega}/\omega \approx 1 + \frac{1}{24}p^2$. The error $\frac{1}{24}p^2$ leads to wrongly displaced peaks of the numerical solution, and the error in peak location grows linearly with time (see Exercise 2).

5 Alternative schemes based on 1st-order equations

A standard technique for solving second-order ODEs is to rewrite them as a system of first-order ODEs and then choose a solution strategy from the vast collection of methods for first-order ODE systems. Given the second-order ODE problem

$$u'' + \omega^2 u = 0, \quad u(0) = I,\ u'(0) = 0,$$

we introduce the auxiliary variable $v = u'$ and express the ODE problem in terms of first-order derivatives of u and v:

$$u' = v, \tag{24}$$
$$v' = -\omega^2 u\,. \tag{25}$$

The initial conditions become $u(0) = I$ and $v(0) = 0$.

5.1 The Forward Euler scheme

A Forward Euler approximation to our 2×2 system of ODEs (24)-(25) becomes

$$[D_t^+ u = v]^n, [D_t^+ v = -\omega^2 u]^n, \tag{26}$$

or written out,

$$u^{n+1} = u^n + \Delta t v^n, \tag{27}$$
$$v^{n+1} = v^n - \Delta t \omega^2 u^n\,. \tag{28}$$

Let us briefly compare this Forward Euler method with the centered difference scheme for the second-order differential equation. We have from (27) and (28) applied at levels n and $n-1$ that

$$u^{n+1} = u^n + \Delta t v^n = u^n + \Delta t(v^{n-1} - \Delta t \omega^2 u^{n-1}\,.$$

Since from (27)

$$v^{n-1} = \frac{1}{\Delta t}(u^n - u^{n-1}),$$

it follows that

$$u^{n+1} = 2u^n - u^{n-1} - \Delta t^2 \omega^2 u^{n-1},$$

which is very close to the centered difference scheme, but the last term is evaluated at t_{n-1} instead of t_n. Dividing by Δt^2, the left-hand side is an approximation to u'' at t_n, while the right-hand side is sampled at t_{n-1}. All terms should be sampled at the same mesh point, so using $\omega^2 u^{n-1}$ instead of $\omega^2 u^n$ is an inconsistency in the scheme. This inconsistency turns out to be rather crucial for the accuracy of the Forward Euler method applied to vibration problems.

5.2 The Backward Euler scheme

A Backward Euler approximation the ODE system is equally easy to write up in the operator notation:

$$[D_t^- u = v]^{n+1}, \tag{29}$$
$$[D_t^- v = -\omega u]^{n+1}\,. \tag{30}$$

This becomes a coupled system for u^{n+1} and v^{n+1}:

$$u^{n+1} - \Delta t v^{n+1} = u^n, \tag{31}$$
$$v^{n+1} + \Delta t \omega^2 u^{n+1} = v^n\,. \tag{32}$$

We can compare (31)-(32) with the centered scheme (7) for the second-order differential equation. To this end, we eliminate v^{n+1} in (31) using (32) solved with respect to v^{n+1}. Thereafter, we eliminate v^n using (31) solved with respect to v^{n+1} and replacing $n+1$ by n. The resulting equation involving only u^{n+1}, u^n, and u^{n-1} can be ordered as

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} = -\omega^2 u^{n+1},$$

which has almost the same form as the centered scheme for the second-order differential equation, but the right-hand side is evaluated at u^{n+1} and not u^n. This inconsistent sampling of terms has a dramatic effect on the numerical solution.

5.3 The Crank-Nicolson scheme

The Crank-Nicolson scheme takes this form in the operator notation:

$$[D_t u = \overline{v}^t]^{n+\frac{1}{2}}, \tag{33}$$
$$[D_t v = -\omega \overline{u}^t]^{n+\frac{1}{2}}\,. \tag{34}$$

Writing the equations out shows that this is also a coupled system:

$$u^{n+1} - \frac{1}{2}\Delta t v^{n+1} = u^n + \frac{1}{2}\Delta t v^n, \tag{35}$$
$$v^{n+1} + \frac{1}{2}\Delta t \omega^2 u^{n+1} = v^n - \frac{1}{2}\Delta t \omega^2 u^n\,. \tag{36}$$

To see the nature of this approximation, and that it is actually very promising, we write the equations as follows

$$u^{n+1} - u^n = \frac{1}{2}\Delta t(v^{n+1} + v^n), \tag{37}$$

$$v^{n+1} = v^n - \frac{1}{2}\Delta t(u^{n+1} + u^n), \tag{38}$$

and add the latter at the previous time level as well:

$$v^n = v^{n-1} - \frac{1}{2}\Delta t(u^n + u^{n-1}) \tag{39}$$

We can also rewrite (5.3) at the previous time level as

$$v^{n+1} + v^n = \frac{2}{\Delta t}(u^{n+1} - u^n)\,. \tag{40}$$

Inserting (38) for v^{n+1} in (ref) and (40) for v^n in (ref) yields after some reordering:

$$u^{n+1} - n^n = \frac{1}{2}(-\frac{1}{2}\Delta t\omega^2(u^{n+1} + 2u^n + u^{n-1}) + v^+v^{n-1})\,.$$

Now, $v^n + v^{n-1}$ can be eliminated by means of (40). The result becomes

$$u^{n+1} - 2u^n + u^{n-1} = \Delta t^2\omega^2\frac{1}{4}(u^{n+1} + 2u^n + u^{n-1})\,. \tag{41}$$

We have that

$$\frac{1}{4}(u^{n+1} + 2u^n + u^{n-1}) \approx u^n + \mathcal{O}(\Delta t^2),$$

meaning that (41) is an approximation to the centered scheme (7) for the second-order ODE where the sampling error in the term $\Delta t^2\omega^2u^n$ is of the same order as the approximation errors in the finite differences, i.e., $\mathcal{O}(\Delta t^2)$. The Crank-Nicolson scheme written as (41) therefore has consistent sampling of all terms at the same time point t_n. The implication is a much better method than the Forward and Backward Euler schemes.

5.4 Comparison of schemes

We can easily compare methods like the ones above (and many more!) with the aid of the Odespy[11] package. Below is a sketch of the code.

```
import odespy
import numpy as np

def f(u, t, w=1):
    u, v = u  # u is array of length 2 holding our [u, v]
    return [v, -w**2*u]

def run_solvers_and_plot(solvers, timesteps_per_period=20,
                         num_periods=1, I=1, w=2*np.pi):
    P = 2*np.pi/w  # duration of one period
    dt = P/timesteps_per_period
    Nt = num_periods*timesteps_per_period
    T = Nt*dt
```

[11] https://github.com/hplgit/odespy


```
    t_mesh = np.linspace(0, T, Nt+1)

    legends = []
    for solver in solvers:
        solver.set(f_kwargs={'w': w})
        solver.set_initial_condition([I, 0])
        u, t = solver.solve(t_mesh)
```

There is quite some more code dealing with plots also, and we refer to the source file `vib_undamped_odespy.py`[12] for details. Observe that keyword arguments in `f(u,t,w=1)` can be supplied through a solver parameter `f_kwargs` (dictionary of additional keyword arguments to `f`).

Specification of the Forward Euler, Backward Euler, and Crank-Nicolson schemes is done like this:

```
solvers = [
    odespy.ForwardEuler(f),
    # Implicit methods must use Newton solver to converge
    odespy.BackwardEuler(f, nonlinear_solver='Newton'),
    odespy.CrankNicolson(f, nonlinear_solver='Newton'),
    ]
```

The `vib_undamped_odespy.py` program makes two plots of the computed solutions with the various methods in the `solvers` list: one plot with $u(t)$ versus t, and one *phase plane plot* where v is plotted against u. That is, the phase plane plot is the curve $(u(t), v(t))$ parameterized by t. Analytically, $u = I\cos(\omega t)$ and $v = u' = -\omega I\sin(\omega t)$. The exact curve $(u(t), v(t))$ is therefore an ellipse, which often looks like a circle in a plot if the axes are automatically scaled. The important feature, however, is that exact curve $(u(t), v(t))$ is closed and repeats itself for every period. Not all numerical schemes are capable of doing that, meaning that the amplitude instead shrinks or grows with time.

Figure 6 show the results. Note that Odespy applies the label MidpointImplicit for what we have specified as `CrankNicolson` in the code (`CrankNicolson` is just a synonym for class `MidpointImplicit` in the Odespy code). The Forward Euler scheme in Figure 6 has a pronounced spiral curve, pointing to the fact that the amplitude steadily grows, which is also evident in Figure 7. The Backward Euler scheme has a similar feature, except that the spriral goes inward and the amplitude is significantly damped. The changing amplitude and the sprial form decreases with decreasing time step. The Crank-Nicolson scheme looks much more accurate. In fact, these plots tell that the Forward and Backward Euler schemes are not suitable for solving our ODEs with oscillating solutions.

5.5 Runge-Kutta methods

We may run two popular standard methods for first-order ODEs, the 2nd- and 4th-order Runge-Kutta methods, to see how they perform. Figures 8 and 9 show the solutions with larger Δt values than what was used in the previous two plots.

The visual impression is that the 4th-order Runge-Kutta method is very accurate, under all circumstances in these tests, while the 2nd-order scheme suffers from amplitude errors unless the time step is very small.

The corresponding results for the Crank-Nicolson scheme are shown in Figure 10. It is clear that the Crank-Nicolson scheme outperforms the 2nd-order Runge-Kutta method. Both schemes have the same order of accuracy $\mathcal{O}(\Delta t^2)$, but their differences in the accuracy that matters in a real physical application is very clearly pronounced in this example. Exercise 12 invites you to investigate how the amplitude is computed by a series of famous methods for first-order ODEs.

[12] http://tinyurl.com/nm5587k/vib/vib_undamped_odespy.py

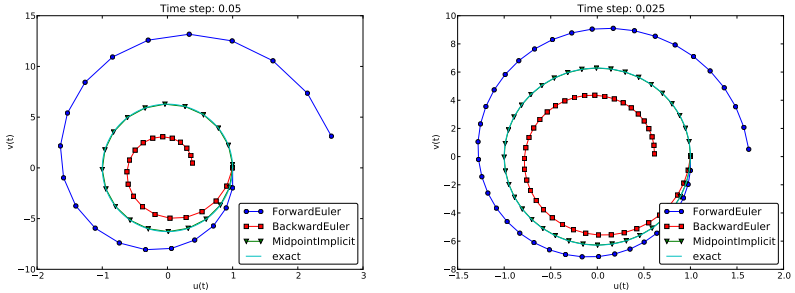


Figure 6: Comparison of classical schemes in the phase plane for two time step values.

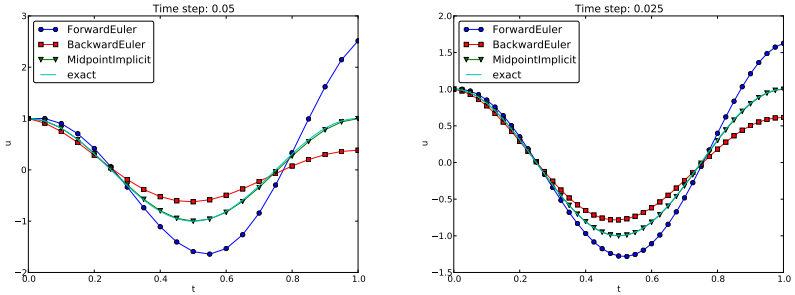


Figure 7: Comparison of solution curves for classical schemes.

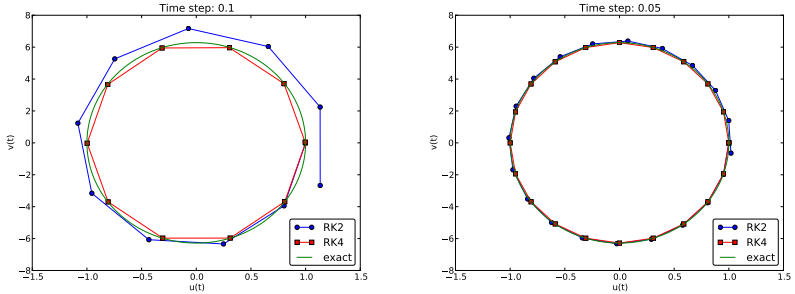


Figure 8: Comparison of Runge-Kutta schemes in the phase plane.

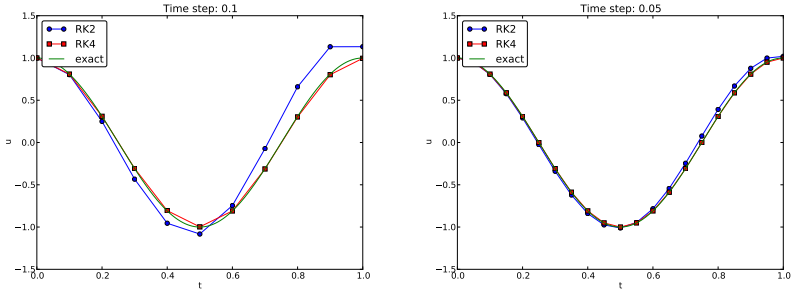


Figure 9: Comparison of Runge-Kutta schemes.

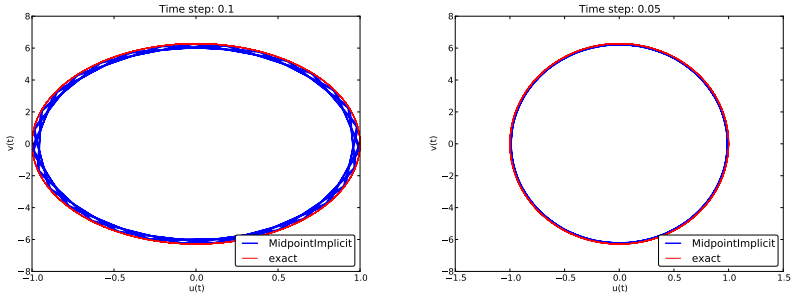


Figure 10: Long-time behavior of the Crank-Nicolson scheme in the phase plane.

5.6 Analysis of the Forward Euler scheme

We may try to find exact solutions of the discrete equations (27)-(28) in the Forward Euler method. An "ansatz" is

$$u^n = IA^n,$$
$$v^n = qIA^n,$$

where q and A are unknown numbers. We could have used a complex exponential form $e^{i\tilde{\omega}n\Delta t}$ since we get oscillatory form, but the oscillations grow in the Forward Euler method, so the numerical frequency $\tilde{\omega}$ will be complex anyway (producing an exponentially growing amplitude). Therefore, it is easier to just work with potentially complex A and q as introduced above.

The Forward Euler scheme leads to

$$A = 1 + \Delta t q,$$
$$A = 1 - \Delta t \omega^2 q^{-1}.$$

We can easily eliminate A, get $q^2 + \omega^2 = 0$, and solve for

$$q = \pm i\omega,$$

which gives

$$A = 1 \pm \Delta t i\omega\,.$$

We shall take the real part of A^n as the solution. The two values of A are complex conjugates, and the real part of A^n will be the same for both roots. This is easy to realize if we rewrite the complex numbers in polar form, which is also convenient for further analysis and understanding. The polar form $re^{i\theta}$ of a complex number $x + iy$ has $r = \sqrt{x^2 + y^2}$ and $\theta = \tan^{-1}(y/x)$. Hence, the polar form of the two values for A become

$$1 \pm \Delta t i\omega = \sqrt{1 + \omega^2\Delta t^2}e^{\pm i\tan^{-1}(\omega\Delta t)}\,.$$

Now it is very easy to compute A^n:

$$(1 \pm \Delta t i\omega)^n = (1 + \omega^2\Delta t^2)^{n/2}e^{\pm ni\tan^{-1}(\omega\Delta t)}\,.$$

Since $\cos(\theta n) = \cos(-\theta n)$, the real part of the two numbers become the same. We therefore continue with the solution that has the plus sign.

The general solution is $u^n = CA^n$, where C is a constant determined from the initial condition: $u^0 = C = I$. We have $u^n = IA^n$ and $v^n = qIA^n$. The final solutions are just the real part of the expressions in polar form:

$$u^n = I(1 + \omega^2\Delta t^2)^{n/2}\cos(n\tan^{-1}(\omega\Delta t)), \tag{42}$$

$$v^n = -\omega I(1 + \omega^2\Delta t^2)^{n/2}\sin(n\tan^{-1}(\omega\Delta t))\,. \tag{43}$$

The expression $(1 + \omega^2\Delta t^2)^{n/2}$ causes growth of the amplitude, since a number greater than one is raised to a positive exponent $n/2$. We can develop a series expression to better understand the formula for the amplitude. Introducing $p = \omega\Delta t$ as the key variable and using `sympy` gives

```
>>> from sympy import *
>>> p = symbols('p', real=True)
>>> n = symbols('n', integer=True, positive=True)
>>> amplitude = (1 + p**2)**(n/2)
>>> amplitude.series(p, 0, 4)
1 + n*p**2/2 + O(p**4)
```

The amplitude goes like $1 + \frac{1}{2}n\omega^2\Delta t^2$, clearly growing linearly in time (with n).

We can also investigate the error in the angular frequency by a series expansion:

```
>>> n*atan(p).series(p, 0, 4)
n*(p - p**3/3 + O(p**4))
```

This means that the solution for u^n can be written as

$$u^n = (1 + \frac{1}{2}n\omega^2\Delta t^2 + \mathcal{O}(()\Delta t^4))\cos\left(\omega t - \frac{1}{3}\omega t\Delta t^2 + \mathcal{O}(\Delta t^4)\right)\,.$$

The error in the angular frequency is of the same order as in the scheme (7) for the second-order ODE, but error in the amplitude is severe.

6 Energy considerations

The observations of various methods in the previous section can be better interpreted if we compute a quantity reflecting the total *energy of the system*. It turns out that this quantity,

$$E(t) = \frac{1}{2}(u')^2 + \frac{1}{2}\omega^2u^2,$$

is *constant* for all t. Checking that $E(t)$ really remains constant brings evidence that the numerical computations are sound. It turns out that E is proportional to the mechanical energy in the system. Conservation of energy is much used to check numerical simulations.

6.1 Derivation of the energy expression

We start out with multiplying

$$u'' + \omega^2u = 0,$$

by u' and integrating from 0 to T:

$$\int_0^T u''u'dt + \int_0^T \omega^2uu'dt = 0\,.$$

Observing that

$$u''u' = \frac{d}{dt}\frac{1}{2}(u')^2, \quad uu' = \frac{d}{dt}\frac{1}{2}u^2,$$

we get

$$\int_0^T (\frac{d}{dt}\frac{1}{2}(u')^2 + \frac{d}{dt}\frac{1}{2}\omega^2u^2)dt = E(T) - E(0) = 0,$$

where we have introduced

$$E(t) = \frac{1}{2}(u')^2 + \frac{1}{2}\omega^2u^2\,. \tag{44}$$

The important result from this derivation is that the total energy is constant:

$$E(t) = E(0)\,.$$

$E(t)$ is closely related to the system's energy.

The quantity $E(t)$ derived above is physically not the mechanical energy of a vibrating mechanical system, but the energy per unit mass. To see this, we start with Newton's second law $F = ma$ (F is the sum of forces, m is the mass of the system, and a is the acceleration). The displacement u is related to a through $a = u''$. With a spring force as the only force we have $F = -ku$, where k is a spring constant measuring the stiffness of the spring. Newton's second law then implies the differential equation

$$-ku = mu'' \quad \Rightarrow mu'' + ku = 0\,.$$

This equation of motion can be turned into an energy balance equation by finding the work done by each term during a time interval $[0, T]$. To this end, we multiply the equation by $du = u'dt$ and integrate:

$$\int_0^T muu'dt + \int_0^T kuu'dt = 0\,.$$

The result is

$$\tilde{E}(t) = E_k(t) + E_p(t) = 0,$$

where

$$E_k(t) = \frac{1}{2}mv^2, \quad v = u', \tag{45}$$

is the *kinetic energy* of the system, and

$$E_p(t) = \frac{1}{2}ku^2 \tag{46}$$

is the *potential energy*. The sum $\tilde{E}(t)$ is the total mechanical energy. The derivation demonstrates the famous energy principle that, under the right physical circumstances, any change in the kinetic energy is due to a change in potential energy and vice versa. (This principle breaks down when we introduce damping in system, as we do in Section 8.)

The equation $mu'' + ku = 0$ can be divided by m and written as $u'' + \omega^2 u = 0$ for $\omega = \sqrt{k/m}$. The energy expression $E(t) = \frac{1}{2}(u')^2 + \frac{1}{2}\omega^2 u^2$ derived earlier is then $\tilde{E}(t)/m$, i.e., mechanical energy per unit mass.

Energy of the exact solution. Analytically, we have $u(t) = I\cos\omega t$, if $u(0) = I$ and $u'(0) = 0$, so we can easily check that the energy evolution and confirm that $E(t)$ is constant:

$$E(t) = \frac{1}{2}I^2(-\omega\sin\omega t)^2 + \frac{1}{2}\omega^2 I^2\cos^2\omega t = \frac{1}{2}\omega^2(\sin^2\omega t + \cos^2\omega t) = \frac{1}{2}\omega^2\,.$$

6.2 An error measure based on energy

The constant energy is well expressed by its initial value $E(0)$, so that the error in mechanical energy can be computed as a mesh function by

$$e_E^n = \frac{1}{2}\left(\frac{u^{n+1} - u^{n-1}}{2\Delta t}\right)^2 + \frac{1}{2}\omega^2(u^n)^2 - E(0), \quad n = 1, \ldots, N_t - 1, \tag{47}$$

where

$$E(0) = \frac{1}{2}V^2 + \frac{1}{2}\omega^2 I^2,$$

if $u(0) = I$ and $u'(0) = V$. Note that we have used a centered approximation to u': $\boldsymbol{u}'(t_n) \approx [D_{2t}u]^n$.

A useful norm of the mesh function e_E^n for the discrete mechanical energy can be the maximum absolute value of e_E^n:

$$||e_E^n||_{\ell^\infty} = \max_{1\le n<N_t} |e_E^n|\,.$$

Alternatively, we can compute other norms involving integration over all mesh points, but we are often interested in worst case deviation of the energy, and then the maximum value is of particular relevance.

A vectorized Python implementation takes the form

```
# import numpy as np and compute u, t
dt = t[1]-t[0]
E = 0.5*((u[2:] - u[:-2])/(2*dt))**2 + 0.5*w**2*u[1:-1]**2
E0 = 0.5*V**2 + 0.5**w**2*I**2
e_E = E - E0
e_E_norm = np.abs(e_E).max()
```

The convergence rates of the quantity `e_E_norm` can be used for verification. The value of `e_E_norm` is also useful for comparing schemes through their ability to preserve energy. Below is a table demonstrating the error in total energy for various schemes. We clearly see that the Crank-Nicolson and 4th-order Runge-Kutta schemes are superior to the 2nd-order Runge-Kutta method and better compared to the Forward and Backward Euler schemes.

Method	T	Δt	$\max\|e_E^n\|$
Forward Euler	1	0.05	$1.113 \cdot 10^{2}$
Forward Euler	1	0.025	$3.312 \cdot 10^{1}$
Backward Euler	1	0.05	$1.683 \cdot 10^{1}$
Backward Euler	1	0.025	$1.231 \cdot 10^{1}$
Runge-Kutta 2nd-order	1	0.1	8.401
Runge-Kutta 2nd-order	1	0.05	$9.637 \cdot 10^{-1}$
Crank-Nicolson	1	0.05	$9.389 \cdot 10^{-1}$
Crank-Nicolson	1	0.025	$2.411 \cdot 10^{-1}$
Runge-Kutta 4th-order	1	0.1	2.387
Runge-Kutta 4th-order	1	0.05	$6.476 \cdot 10^{-1}$
Crank-Nicolson	10	0.1	3.389
Crank-Nicolson	10	0.05	$9.389 \cdot 10^{-1}$
Runge-Kutta 4th-order	10	0.1	3.686
Runge-Kutta 4th-order	10	0.05	$6.928 \cdot 10^{-1}$

7 The Euler-Cromer method

While the 4th-order Runge-Kutta method and a Crank-Nicolson scheme work well for vibration equation modeled as a first-order ODE system, both were inferior to the straightforward centered difference scheme for the second-order equation $u'' + \omega^2 u = 0$. However, there is a similarly successful scheme available for the first-order system $u' = v$, $v' = -\omega^2 u$, to be presented next.

7.1 Forward-backward discretization

The idea is to apply a Forward Euler discretization to the first equation and a Backward Euler discretization to the second. In operator notation this is stated as

$$[D_t^+ u = v]^n, \tag{48}$$
$$[D_t^- v = -\omega u]^{n+1}\,. \tag{49}$$

We can write out the formulas and collect the unknowns on the left-hand side:

$$u^{n+1} = u^n + \Delta t v^n, \tag{50}$$
$$v^{n+1} = v^n - \Delta t \omega^2 u^{n+1}\,. \tag{51}$$

We realize that after u^{n+1} has been computed from (50), it may be used directly in (51) to compute v^{n+1}.

In physics, it is more common to update the v equation first, with a forward difference, and thereafter the u equation, with a backward difference that applies the most recently computed v value:

$$v^{n+1} = v^n - \Delta t \omega^2 u^n, \tag{52}$$
$$u^{n+1} = u^n + \Delta t v^{n+1}\,. \tag{53}$$

The advantage of ordering the ODEs as in (52)-(53) becomes evident when consider complicated models. Such models are included if we write our vibration ODE more generally as

$$\ddot{u} + g(u, u', t) = 0\,.$$

We can rewrite this second-order ODE as two first-order ODEs,

$$v' = -g(u, v, t),$$
$$u' = v\,.$$

This rewrite allows the following scheme to be used:

$$v^{n+1} = v^n - \Delta t\, g(u^n, v^n, t),$$
$$u^{n+1} = u^n + \Delta t\, v^{n+1}\,.$$

We realize that the first update works well with any g since old values u^n and v^n are used. Switching the equations would demand $u^n{+}1$ and v^{n+1} values in g.

The scheme (52)-(53) goes under several names: forward-backward scheme, semi-implicit Euler method[13], semi-explicit Euler, symplectic Euler, Newton-Störmer-Verlet, and Euler-Cromer. We shall stick to the latter name. Since both time discretizations are based on first-order difference approximation, one may think that the scheme is only of first-order, but this is not true: the use of a forward and then a backward difference make errors cancel so that the overall error in the scheme is $\mathcal{O}(\Delta t^2)$. This is explained below.

[13] http://en.wikipedia.org/wiki/Semi-implicit_Euler_method

7.2 Equivalence with the scheme for the second-order ODE

We may eliminate the v^n variable from (50)-(51) or (52)-(53). The v^{n+1} term in (52) can be eliminated from (53):

$$u^{n+1} = u^n + \Delta t(v^n - \omega^2 \Delta t^2 u^n)\,. \tag{54}$$

The v^n quantity can be expressed by u^n and u^{n-1} using (53):

$$v^n = \frac{u^n - u^{n-1}}{\Delta t},$$

and when this is inserted in (54) we get

$$u^{n+1} = 2u^n - u^{n-1} - \Delta t^2 \omega^2 u^n, \tag{55}$$

which is nothing but the centered scheme (7)! The two seemingly different numerical methods are mathematically equivalent. Consequently, the previous analysis of (7) also applies to the Euler-Cromer method. In particular, the amplitude is constant, given that the stability criterion is fulfilled, but there is always an angular frequency error (19). Exercise 17 gives guidance on how to derive the exact discrete solution of the two equations in the Euler-Cromer method.

Although the Euler-Cromer scheme and the method (7) are equivalent, there could be differences in the way they handle the initial conditions. Let is look into this topic. The initial condition $u' = 0$ means $u' = v = 0$. From (53) we get $v^1 = -\omega^2 u^0$ and $u^1 = u^0 - \omega^2 \Delta t^2 u^0$. When using a centered approximation of $u'(0) = 0$ combined with the discretization (7) of the second-order ODE, we get $u^1 = u^0 - \frac{1}{2}\omega^2 \Delta t^2 u^0$. The difference is $\frac{1}{2}\omega^2 \Delta t^2 u^0$, which is of second order in Δt, seemingly consistent with the overall error in the scheme for the differential equation model.

A different view can also be taken. If we approximate $u'(0) = 0$ by a backward difference, $(u^0 - u^{-1})/\Delta t = 0$, we get $u^{-1} = u^0$, and when combined with (7), it results in $u^1 = u^0 - \omega^2 \Delta t^2 u^0$. This means that the Euler-Cromer method based on (53)-(52) corresponds to using only a first-order approximation to the initial condition in the method from Section 1.2.

Correspondingly, using the formulation (50)-(51) with $v^n = 0$ leads to $u^1 = u^0$, which can be interpreted as using a forward difference approximation for the initial condition $u'(0) = 0$. Both Euler-Cromer formulations lead to slightly different values for u^1 compared to the method in Section 1.2. The error is $\frac{1}{2}\omega^2 \Delta t^2 u^0$ and of the same order as the overall scheme.

7.3 Implementation

The function below, found in `vib_EulerCromer.py`[14] implements the Euler-Cromer scheme (52)-(53):

```
import numpy as np

def solver(I, w, dt, T):
    """
    Solve v' = - w**2*u, u'=v for t in (0,T], u(0)=I and v(0)=0,
    by an Euler-Cromer method.
    """
    dt = float(dt)
    Nt = int(round(T/dt))
    u = np.zeros(Nt+1)
    v = np.zeros(Nt+1)
    t = np.linspace(0, Nt*dt, Nt+1)
```

[14] http://tinyurl.com/nm5587k/vib/vib_EulerCromer.py

```
    v[0] = 0
    u[0] = I
    for n in range(0, Nt):
        v[n+1] = v[n] - dt*w**2*u[n]
        u[n+1] = u[n] + dt*v[n+1]
    return u, v, t
```

Since the Euler-Cromer scheme is equivalent to the finite difference method for the second-order ODE $u'' + \omega^2 u = 0$ (see Section 7.2), the performance of the above `solver` function is the same as for the `solver` function in Section 2. The only difference is the formula for the first time step, as discussed above. This deviation in the Euler-Cromer scheme means that the discrete solution listed in Section 4.2 is not a solution of the Euler-Cromer scheme!

To verify the implementation of the Euler-Cromer method we can adjust `v[1]` so that the computer-generated values can be compared with the formula (20) from in Section 4.2. This adjustment is done in an alternative solver function, `solver_ic_fix` in `vib_EulerCromer.py`. Since we now have an exact solution of the discrete equations available, we can write a test function `test_solver` for checking the equality of computed values with the formula (20):

```
def test_solver():
    """
    Test solver with fixed initial condition against
    equivalent scheme for the 2nd-order ODE u'' + u = 0.
    """
    I = 1.2; w = 2.0; T = 5
    dt = 2/w  # longest possible time step
    u, v, t = solver_ic_fix(I, w, dt, T)
    from vib_undamped import solver as solver2  # 2nd-order ODE
    u2, t2 = solver2(I, w, dt, T)
    error = np.abs(u - u2).max()
    tol = 1E-14
    assert error < tol
```

Another function, `demo`, visualizes the difference between Euler-Cromer scheme and the scheme (7) for the second-oder ODE, arising from the mismatch in the first time level.

7.4 The velocity Verlet algorithm

Another very popular algorithm for vibration problems $u'' + \omega^2 u = 0$ can be derived as follows. First, we step u forward from t_n to t_{n+1} using a three-term Taylor series,

$$u(t_{n+1}) = u(t_n) + u'(t_n)\Delta t + \frac{1}{2}u''(t_n)\Delta t^2\,.$$

Using $u' = v$ and $u'' = -\omega^2 u$, we get the updating formula

$$u^{n+1} = u^n + v^n\Delta t - \frac{1}{2}\Delta^2\omega^2 u^n\,.$$

Second, the first-order equation for v,

$$v' = -\omega^2 u,$$

is discretized by a centered difference in a Crank-Nicolson fashion at $t_{n+\frac{1}{2}}$:

$$\frac{v^{n+1} - v^n}{\Delta t} = -\omega^2\frac{1}{2}(u^n + u^{n+1})\,.$$

To summarize, we have the scheme

$$u^{n+1} = u^n + v^n\Delta t - \frac{1}{2}\Delta^2\omega^2 u^n \tag{56}$$

$$v^{n+1} = v^n - \frac{1}{2}\Delta t\omega^2(u^n + u^{n+1}), \tag{57}$$

known as the *velocity Verlet* algorithm. Observe that this scheme is explicit since u^{n+1} in (57) is already computed from (56).

The algorithm can be straightforwardly implemented as shown below (the code appears in the file `vib_undamped_velocity_Verlet.py`[15]).

```
from vib_undamped import convergence_rates, main

def solver(I, w, dt, T, return_v=False):
    """
    Solve u'=v, v'=-w**2*u for t in (0,T], u(0)=I and v(0)=0,
    by the velocity Verlet method with time step dt.
    """
    dt = float(dt)
    Nt = int(round(T/dt))
    u = np.zeros(Nt+1)
    v = np.zeros(Nt+1)
    t = np.linspace(0, Nt*dt, Nt+1)

    u[0] = I
    v[0] = 0
    for n in range(Nt):
        u[n+1] = u[n] + v[n]*dt - 0.5*dt**2*w**2*u[n]
        v[n+1] = v[n] - 0.5*dt*w**2*(u[n] + u[n+1])
    if return_v:
        return u, v, t
    else:
        # Return just u and t as in the vib_undamped.py's solver
        return u, t
```

We provide the option that this `solver` function returns the same data as the `solver` function from Section 2.1 (if `return_v` is `False`), but we may return `v` along with `u` and `t`.

The error in the Taylor series expansion behind (56) is $\mathcal{O}(\Delta t^3)$, while the error in the central difference for v is $\mathcal{O}(\Delta t^2)$. The overall error is then no better than $\mathcal{O}(\Delta t^2)$, which can be verified empirically using the `convergence_rates` function from 2.2:

```
>>> import vib_undamped_velocity_Verlet as m
>>> m.convergence_rates(4, solver_function=m.solver)
[2.0036366687367346, 2.000949728124835, 2.000240105995295]
```

8 Generalization: damping, nonlinear spring, and external excitation

We shall now generalize the simple model problem from Section 1 to include a possibly nonlinear damping term $f(u')$, a possibly nonlinear spring (or restoring) force $s(u)$, and some external excitation $F(t)$:

$$mu'' + f(u') + s(u) = F(t), \quad u(0) = I,\ u'(0) = V,\ t \in (0, T]\,. \tag{58}$$

[15] `http://tinyurl.com/nm5587k/vib/vib_undamped_velocity_Verlet.py`

We have also included a possibly nonzero initial value of $u'(0)$. The parameters m, $f(u')$, $s(u)$, $F(t)$, I, V, and T are input data.

There are two main types of damping (friction) forces: linear $f(u') = bu$, or quadratic $f(u') = bu'|u'|$. Spring systems often feature linear damping, while air resistance usually gives rise to quadratic damping. Spring forces are often linear: $s(u) = cu$, but nonlinear versions are also common, the most famous is the gravity force on a pendulum that acts as a spring with $s(u) \sim \sin(u)$.

8.1 A centered scheme for linear damping

Sampling (58) at a mesh point t_n, replacing $u''(t_n)$ by $[D_tD_tu]^n$, and $u'(t_n)$ by $[D_{2t}u]^n$ results in the discretization

$$[mD_tD_tu + f(D_{2t}u) + s(u) = F]^n, \tag{59}$$

which written out means

$$m\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} + f(\frac{u^{n+1} - u^{n-1}}{2\Delta t}) + s(u^n) = F^n, \tag{60}$$

where F^n as usual means $F(t)$ evaluated at $t = t_n$. Solving (60) with respect to the unknown u^{n+1} gives a problem: the u^{n+1} inside the f function makes the equation *nonlinear* unless $f(u')$ is a linear function, $f(u') = bu'$. For now we shall assume that f is linear in u'. Then

$$m\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} + b\frac{u^{n+1} - u^{n-1}}{2\Delta t} + s(u^n) = F^n, \tag{61}$$

which gives an explicit formula for u at each new time level:

$$u^{n+1} = (2mu^n + (\frac{b}{2}\Delta t - m)u^{n-1} + \Delta t^2(F^n - s(u^n)))(m + \frac{b}{2}\Delta t)^{-1}\,. \tag{62}$$

For the first time step we need to discretize $u'(0) = V$ as $[D_{2t}u = V]^0$ and combine with (62) for $n = 0$. The discretized initial condition leads to

$$u^{-1} = u^1 - 2\Delta t V, \tag{63}$$

which inserted in (62) for $n = 0$ gives an equation that can be solved for u^1:

$$u^1 = u^0 + \Delta t\, V + \frac{\Delta t^2}{2m}(-bV - s(u^0) + F^0)\,. \tag{64}$$

8.2 A centered scheme for quadratic damping

When $f(u') = bu'|u'|$, we get a quadratic equation for u^{n+1} in (60). This equation can be straightforwardly solved by the well-known formula for the roots of a quadratic equation. However, we can also avoid the nonlinearity by introducing an approximation with an error of order no higher than what we already have from replacing derivatives with finite differences.

We start with (58) and only replace u'' by D_tD_tu, resulting in

$$[mD_tD_tu + bu'|u'| + s(u) = F]^n\,. \tag{65}$$

Here, $u'|u'|$ is to be computed at time t_n. The idea is now to introduce a *geometric mean*, defined by

$$(w^2)^n \approx w^{n-\frac{1}{2}}w^{n+\frac{1}{2}},$$

for some quantity w depending on time. The error in the geometric mean approximation is $\mathcal{O}(\Delta t^2)$, the same as in the approximation $u'' \approx D_tD_tu$. With $w = u'$ it follows that

$$[u'|u'|]^n \approx u'(t_{n+\frac{1}{2}})|u'(t_{n-\frac{1}{2}})|\,.$$

The next step is to approximate u' at $t_{n\pm 1/2}$, and fortunately a centered difference fits perfectly into the formulas since it involves u values at the mesh points only. With the approximations

$$u'(t_{n+1/2}) \approx [D_tu]^{n+\frac{1}{2}}, \quad u'(t_{n-1/2}) \approx [D_tu]^{n-\frac{1}{2}}, \tag{66}$$

we get

$$[u'|u'|]^n \approx [D_tu]^{n+\frac{1}{2}}|[D_tu]^{n-\frac{1}{2}}| = \frac{u^{n+1} - u^n}{\Delta t}\frac{|u^n - u^{n-1}|}{\Delta t}\,. \tag{67}$$

The counterpart to (60) is then

$$m\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} + b\frac{u^{n+1} - u^n}{\Delta t}\frac{|u^n - u^{n-1}|}{\Delta t} + s(u^n) = F^n, \tag{68}$$

which is linear in the unknown u^{n+1}. Therefore, we can easily solve (68) with respect to u^{n+1} and achieve the explicit updating formula

$$u^{n+1} = \left(m + b|u^n - u^{n-1}|\right)^{-1} \times \left(2mu^n - mu^{n-1} + bu^n|u^n - u^{n-1}| + \Delta t^2(F^n - s(u^n))\right)\,. \tag{69}$$

In the derivation of a special equation for the first time step we run into some trouble: inserting (63) in (69) for $n = 0$ results in a complicated nonlinear equation for u^1. By thinking differently about the problem we can easily get away with the nonlinearity again. We have for $n = 0$ that $b[u'|u'|]^0 = bV|V|$. Using this value in (65) gives

$$[mD_tD_tu + bV|V| + s(u) = F]^0\,. \tag{70}$$

Writing this equation out and using (63) results in the special equation for the first time step:

$$u^1 = u^0 + \Delta t V + \frac{\Delta t^2}{2m}\left(-bV|V| - s(u^0) + F^0\right)\,. \tag{71}$$

8.3 A forward-backward discretization of the quadratic damping term

The previous section first proposed to discretize the quadratic damping term $|u'|u'$ using centered differences: $[|D_{2t}|D_{2t}u]^n$. As this gives rise to a nonlinearity in u^{n+1}, it was instead proposed to use a geometric mean combined with centered differences. But there are other alternatives. To get rid of the nonlinearity in $[|D_{2t}|D_{2t}u]^n$, one can think differently: apply a backward difference to $|u'|$, such that the term involves known values, and apply a forward difference to u' to make the term linear in the unknown u^{n+1}. With mathematics,

$$[\beta|u'|u']^n \approx \beta|[D_t^-u]^n|[D_t^+u]^n = \beta\left|\frac{u^n - u^{n-1}}{\Delta t}\right|\frac{u^{n+1} - u^n}{\Delta t}\,. \tag{72}$$

The forward and backward differences have both an error proportional to Δt so one may think the discretization above leads to a first-order scheme. However, by looking at the formulas, we realize that the forward-backward differences in (72) result in exactly the same scheme as in (68) where we used a geometric mean and centered differences and committed errors of size $\mathcal{O}(\Delta t^2)$. Therefore, the forward-backward differences in (72) act in a symmetric way and actually produce a second-order accurate discretization of the quadratic damping term.

8.4 Implementation

The algorithm arising from the methods in Sections 8.1 and 8.2 is very similar to the undamped case in Section 1.2. The difference is basically a question of different formulas for u^1 and u^{n+1}. This is actually quite remarkable. The equation (58) is normally impossible to solve by pen and paper, but possible for some special choices of F, s, and f. On the contrary, the complexity of the nonlinear generalized model (58) versus the simple undamped model is not a big deal when we solve the problem numerically!

The computational algorithm takes the form

1. $u^0 = I$
2. compute u^1 from (64) if linear damping or (71) if quadratic damping
3. for $n = 1, 2, \ldots, N_t - 1$:
   (a) compute u^{n+1} from (62) if linear damping or (69) if quadratic damping

Modifying the `solver` function for the undamped case is fairly easy, the big difference being many more terms and if tests on the type of damping:

```
def solver(I, V, m, b, s, F, dt, T, damping='linear'):
    """
    Solve m*u'' + f(u') + s(u) = F(t) for t in (0,T],
    u(0)=I and u'(0)=V,
    by a central finite difference method with time step dt.
    If damping is 'linear', f(u')=b*u, while if damping is
    'quadratic', f(u')=b*u'*abs(u').
    F(t) and s(u) are Python functions.
    """
    dt = float(dt); b = float(b); m = float(m) # avoid integer div.
    Nt = int(round(T/dt))
    u = np.zeros(Nt+1)
    t = np.linspace(0, Nt*dt, Nt+1)

    u[0] = I
    if damping == 'linear':
        u[1] = u[0] + dt*V + dt**2/(2*m)*(-b*V - s(u[0]) + F(t[0]))
    elif damping == 'quadratic':
        u[1] = u[0] + dt*V + \
               dt**2/(2*m)*(-b*V*abs(V) - s(u[0]) + F(t[0]))

    for n in range(1, Nt):
        if damping == 'linear':
            u[n+1] = (2*m*u[n] + (b*dt/2 - m)*u[n-1] +
                      dt**2*(F(t[n]) - s(u[n])))/(m + b*dt/2)
        elif damping == 'quadratic':
            u[n+1] = (2*m*u[n] - m*u[n-1] + b*u[n]*abs(u[n] - u[n-1])
                      + dt**2*(F(t[n]) - s(u[n])))/\
                      (m + b*abs(u[n] - u[n-1]))
    return u, t
```


The complete code resides in the file `vib.py`[16].

8.5 Verification

Constant solution. For debugging and initial verification, a constant solution is often very useful. We choose $u_{\mathrm{e}}(t) = I$, which implies $V = 0$. Inserted in the ODE, we get $F(t) = s(I)$ for any choice of f. Since the discrete derivative of a constant vanishes (in particular, $[D_{2t}I]^n = 0$, $[D_tI]^n = 0$, and $[D_tD_tI]^n = 0$), the constant solution also fulfills the discrete equations. The constant should therefore be reproduced to machine precision. The function `test_constant` in `vib.py` implements this test.

Linear solution. Now we choose a linear solution: $u_{\mathrm{e}} = ct + d$. The initial condition $u(0) = I$ implies $d = I$, and $u'(0) = V$ forces c to be V. Inserting $u_{\mathrm{e}} = Vt + I$ in the ODE with linear damping results in

$$0 + bV + s(Vt + I) = F(t),$$

while quadratic damping requires the source term

$$0 + b|V|V + s(Vt + I) = F(t)\,.$$

Since the finite difference approximations used to compute u' all are exact for a linear function, it turns out that the linear u_{e} is also a solution of the discrete equations. Exercise 9 asks you to carry out all the details.

Quadratic solution. Choosing $u_{\mathrm{e}} = bt^2 + Vt + I$, with b arbitrary, fulfills the initial conditions and fits the ODE if F is adjusted properly. The solution also solves the discrete equations with linear damping. However, this quadratic polynomial in t does not fulfill the discrete equations in case of quadratic damping, because the geometric mean used in the approximation of this term introduces an error. Doing Exercise 9 will reveal the details. One can fit F^n in the discrete equations such that the quadratic polynomial is reproduced by the numerical method (to machine precision).

8.6 Visualization

The functions for visualizations differ significantly from those in the undamped case in the `vib_undamped.py` program because, in the present general case, we do not have an exact solution to include in the plots. Moreover, we have no good estimate of the periods of the oscillations as there will be one period determined by the system parameters, essentially the approximate frequency $\sqrt{s'(0)/m}$ for linear s and small damping, and one period dictated by $F(t)$ in case the excitation is periodic. This is, however, nothing that the program can depend on or make use of. Therefore, the user has to specify T and the window width to get a plot that moves with the graph and shows the most recent parts of it in long time simulations.

The `vib.py` code contains several functions for analyzing the time series signal and for visualizing the solutions.

[16] `http://tinyurl.com/nm5587k/vib/vib.py`

8.7 User interface

The `main` function is changed substantially from the `vib_undamped.py` code, since we need to specify the new data c, $s(u)$, and $F(t)$. In addition, we must set T and the plot window width (instead of the number of periods we want to simulate as in `vib_undamped.py`). To figure out whether we can use one plot for the whole time series or if we should follow the most recent part of u, we can use the `plot_empricial_freq_and_amplitude` function's estimate of the number of local maxima. This number is now returned from the function and used in `main` to decide on the visualization technique.

```
def main():
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--I', type=float, default=1.0)
    parser.add_argument('--V', type=float, default=0.0)
    parser.add_argument('--m', type=float, default=1.0)
    parser.add_argument('--c', type=float, default=0.0)
    parser.add_argument('--s', type=str, default='u')
    parser.add_argument('--F', type=str, default='0')
    parser.add_argument('--dt', type=float, default=0.05)
    parser.add_argument('--T', type=float, default=140)
    parser.add_argument('--damping', type=str, default='linear')
    parser.add_argument('--window_width', type=float, default=30)
    parser.add_argument('--savefig', action='store_true')
    a = parser.parse_args()
    from scitools.std import StringFunction
    s = StringFunction(a.s, independent_variable='u')
    F = StringFunction(a.F, independent_variable='t')
    I, V, m, c, dt, T, window_width, savefig, damping = \
       a.I, a.V, a.m, a.c, a.dt, a.T, a.window_width, a.savefig, \
       a.damping

    u, t = solver(I, V, m, c, s, F, dt, T)
    num_periods = empirical_freq_and_amplitude(u, t)
    if num_periods <= 15:
        figure()
        visualize(u, t)
    else:
        visualize_front(u, t, window_width, savefig)
    show()
```

The program `vib.py` contains the above code snippets and can solve the model problem (58). As a demo of `vib.py`, we consider the case $I = 1$, $V = 0$, $m = 1$, $c = 0.03$, $s(u) = \sin(u)$, $F(t) = 3\cos(4t)$, $\Delta t = 0.05$, and $T = 140$. The relevant command to run is

Terminal

```
Terminal> python vib.py --s 'sin(u)' --F '3*cos(4*t)' --c 0.03
```

This results in a moving window following the function[17] on the screen. Figure 11 shows a part of the time series.

8.8 The Euler-Cromer scheme for the generalized model

The ideas of the Euler-Cromer method from Section 7 carry over to the generalized model. We write (58) as two equations for u and $v = u'$. The first equation is taken as the one with v' on the left-hand side:

[17] http://tinyurl.com/opdfafk/pub/mov-vib/vib_generalized_dt0.05/index.html

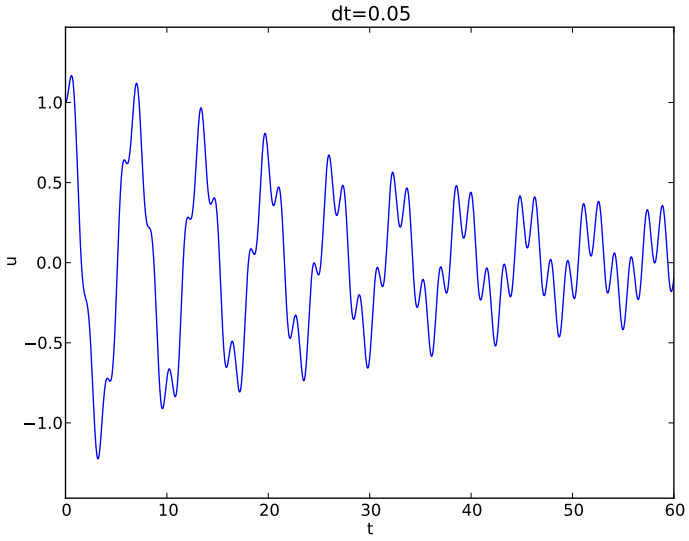


Figure 11: Damped oscillator excited by a sinusoidal function.

$$v' = \frac{1}{m}(F(t) - s(u) - f(v)), \tag{73}$$

$$u' = v\,. \tag{74}$$

The idea is to step (73) forward using a standard Forward Euler method, while we update u from (74) with a Backward Euler method, utilizing the recent, computed v^{n+1} value. In detail,

$$\frac{v^{n+1} - v^n}{\Delta t} = \frac{1}{m}(F(t_n) - s(u^n) - f(v^n)), \tag{75}$$

$$\frac{u^{n+1} - u^n}{\Delta t} = v^{n+1}, \tag{76}$$

resulting in the explicit scheme

$$v^{n+1} = v^n + \Delta t\frac{1}{m}(F(t_n) - s(u^n) - f(v^n)), \tag{77}$$

$$u^{n+1} = u^n + \Delta t\, v^{n+1}\,. \tag{78}$$

We immediately note one very favorable feature of this scheme: all the nonlinearities in $s(u)$ and $f(v)$ are evaluated at a previous time level. This makes the Euler-Cromer method easier to apply and hence much more convenient than the centered scheme for the second-order ODE (58).

The initial conditions are trivially set as

$$v^0 = V, \tag{79}$$

$$u^0 = I\,. \tag{80}$$

9 Exercises and Problems

Problem 1: Use linear/quadratic functions for verification

Consider the ODE problem

$$u'' + \omega^2 u = f(t), \quad u(0) = I,\ u'(0) = V,\ t \in (0, T]\,.$$

Discretize this equation according to $[D_t D_t u + \omega^2 u = f]^n$.

a) Derive the equation for the first time step (u^1).

b) For verification purposes, we use the method of manufactured solutions (MMS) with the choice of $u_{\mathrm{e}}(x,t) = ct + d$. Find restrictions on c and d from the initial conditions. Compute the corresponding source term f by term. Show that $[D_t D_t t]^n = 0$ and use the fact that the $D_t D_t$ operator is linear, $[D_t D_t(ct + d)]^n = c[D_t D_t t]^n + [D_t D_t d]^n = 0$, to show that u_{e} is also a perfect solution of the discrete equations.

c) Use `sympy` to do the symbolic calculations above. Here is a sketch of the program `vib_undamped_verify_mms.py`:

```
import sympy as sym
V, t, I, w, dt = sym.symbols('V t I w dt')  # global symbols
f = None  # global variable for the source term in the ODE

def ode_source_term(u):
    """Return the terms in the ODE that the source term
    must balance, here u'' + w**2*u.
    u is symbolic Python function of t."""
    return sym.diff(u(t), t, t) + w**2*u(t)

def residual_discrete_eq(u):
    """Return the residual of the discrete eq. with u inserted."""
    R = ...
    return sym.simplify(R)

def residual_discrete_eq_step1(u):
    """Return the residual of the discrete eq. at the first
    step with u inserted."""
    R = ...
    return sym.simplify(R)

def DtDt(u, dt):
    """Return 2nd-order finite difference for u_tt.
    u is a symbolic Python function of t.
    """
    return ...

def main(u):
    """
    Given some chosen solution u (as a function of t, implemented
```



```
    as a Python function), use the method of manufactured solutions
    to compute the source term f, and check if u also solves
    the discrete equations.
    """
    print '=== Testing exact solution: %s ===' % u
    print "Initial conditions u(0)=%s, u'(0)=%s:" % \
          (u(t).subs(t, 0), sym.diff(u(t), t).subs(t, 0))

    # Method of manufactured solution requires fitting f
    global f  # source term in the ODE
    f = sym.simplify(ode_lhs(u))

    # Residual in discrete equations (should be 0)
    print 'residual step1:', residual_discrete_eq_step1(u)
    print 'residual:', residual_discrete_eq(u)

def linear():
    main(lambda t: V*t + I)

if __name__ == '__main__':
    linear()
```

Fill in the various functions such that the calls in the `main` function works.

d) The purpose now is to choose a quadratic function $u_{\mathrm{e}} = bt^2 + ct + d$ as exact solution. Extend the `sympy` code above with a function `quadratic` for fitting `f` and checking if the discrete equations are fulfilled. (The function is very similar to `linear`.)

e) Will a polynomial of degree three fulfill the discrete equations?

f) Implement a `solver` function for computing the numerical solution of this problem.

g) Write a nose test for checking that the quadratic solution is computed to correctly (too machine precision, but the round-off errors accumulate and increase with T) by the `solver` function.
Filename: `vib_undamped_verify_mms`.

Exercise 2: Show linear growth of the phase with time

Consider an exact solution $I\cos(\omega t)$ and an approximation $I\cos(\tilde{\omega} t)$. Define the phase error as time lag between the peak I in the exact solution and the corresponding peak in the approximation after m periods of oscillations. Show that this phase error is linear in m. Filename: `vib_phase_error_growth`.

Exercise 3: Improve the accuracy by adjusting the frequency

According to (19), the numerical frequency deviates from the exact frequency by a (dominating) amount $\omega^3\Delta t^2/24 > 0$. Replace the `w` parameter in the algorithm in the `solver` function in `vib_undamped.py` by `w*(1 - (1./24)*w**2*dt**2` and test how this adjustment in the numerical algorithm improves the accuracy (use $\Delta t = 0.1$ and simulate for 80 periods, with and without adjustment of ω). Filename: `vib_adjust_w`.

Exercise 4: See if adaptive methods improve the phase error

Adaptive methods for solving ODEs aim at adjusting Δt such that the error is within a user-prescribed tolerance. Implement the equation $u'' + u = 0$ in the Odespy[18] software. Use the

[18] `https://github.com/hplgit/odespy`

example from Section 3.2.11 in [1]. Run the scheme with a very low tolerance (say 10^{-14}) and for a long time, check the number of time points in the solver's mesh (`len(solver.t_all)`), and compare the phase error with that produced by the simple finite difference method from Section 1.2 with the same number of (equally spaced) mesh points. The question is whether it pays off to use an adaptive solver or if equally many points with a simple method gives about the same accuracy. Filename: `vib_undamped_adaptive`.

Exercise 5: Use a Taylor polynomial to compute u^1

As an alternative to the derivation of (8) for computing u^1, one can use a Taylor polynomial with three terms for u^1:

$$u(t_1) \approx u(0) + u'(0)\Delta t + \frac{1}{2}u''(0)\Delta t^2$$

With $u'' = -\omega^2 u$ and $u'(0) = 0$, show that this method also leads to (8). Generalize the condition on $u'(0)$ to be $u'(0) = V$ and compute u^1 in this case with both methods. Filename: `vib_first_step`.

Exercise 6: Find the minimal resolution of an oscillatory function

Sketch the function on a given mesh which has the highest possible frequency. That is, this oscillatory "cos-like" function has its maxima and minima at every two grid points. Find an expression for the frequency of this function, and use the result to find the largest relevant value of $\omega\Delta t$ when ω is the frequency of an oscillating function and Δt is the mesh spacing. Filename: `vib_largest_wdt`.

Exercise 7: Visualize the accuracy of finite differences for a cosine function

We introduce the error fraction

$$E = \frac{[D_t D_t u]^n}{u''(t_n)}$$

to measure the error in the finite difference approximation $D_t D_t u$ to u''. Compute E for the specific choice of a cosine/sine function of the form $u = \exp(i\omega t)$ and show that

$$E = \left(\frac{2}{\omega\Delta t}\right)^2 \sin^2(\frac{\omega\Delta t}{2}).$$

Plot E as a function of $p = \omega\Delta t$. The relevant values of p are $[0, \pi]$ (see Exercise 6 for why $p > \pi$ does not make sense). The deviation of the curve from unity visualizes the error in the approximation. Also expand E as a Taylor polynomial in p up to fourth degree (use, e.g., `sympy`). Filename: `vib_plot_fd_exp_error`.

Exercise 8: Verify convergence rates of the error in energy

We consider the ODE problem $u'' + \omega^2 u = 0$, $u(0) = I$, $u'(0) = V$, for $t \in (0, T]$. The total energy of the solution $E(t) = \frac{1}{2}(u')^2 + \frac{1}{2}\omega^2 u^2$ should stay constant. The error in energy can be computed as explained in Section 6.

Make a nose test in a file `test_error_conv.py`, where code from `vib_undamped.py` is imported, but the `convergence_rates` and `test_convergence_rates` functions are copied and

modified to also incorporate computations of the error in energy and the convergence rate of this error. The expected rate is 2. Filename: `test_error_conv`.

Exercise 9: Use linear/quadratic functions for verification

This exercise is a generalization of Problem 1 to the extended model problem (58) where the damping term is either linear or quadratic. Solve the various subproblems and see how the results and problem settings change with the generalized ODE in case of linear or quadratic damping. By modifying the code from Problem 1, `sympy` will do most of the work required to analyze the generalized problem. Filename: `vib_verify_mms`.

Exercise 10: Use an exact discrete solution for verification

Write a nose test function in a separate file that employs the exact discrete solution (20) to verify the implementation of the `solver` function in the file `vib_undamped.py`. Filename: `test_vib_undamped_exact_discrete_sol`.

Exercise 11: Use analytical solution for convergence rate tests

The purpose of this exercise is to perform convergence tests of the problem (58) when $s(u) = \omega^2 u$ and $F(t) = A\sin\phi t$. Find the complete analytical solution to the problem in this case (most textbooks on mechanics or ordinary differential equations list the various elements you need to write down the exact solution). Modify the `convergence_rate` function from the `vib_undamped.py` program to perform experiments with the extended model. Verify that the error is of order Δt^2. Filename: `vib_conv_rate`.

Exercise 12: Investigate the amplitude errors of many solvers

Use the program `vib_undamped_odespy.py` from Section 5.4 and the amplitude estimation from the `amplitudes` function in the `vib_undamped.py` file (see Section 3.5) to investigate how well famous methods for 1st-order ODEs can preserve the amplitude of u in undamped oscillations. Test, for example, the 3rd- and 4th-order Runge-Kutta methods (`RK3`, `RK4`), the Crank-Nicolson method (`CrankNicolson`), the 2nd- and 3rd-order Adams-Bashforth methods (`AdamsBashforth2`, `AdamsBashforth3`), and a 2nd-order Backwards scheme (`Backward2Step`). The relevant governing equations are listed in the beginning of Section 5. Filename: `vib_amplitude_errors`.

Exercise 13: Minimize memory usage of a vibration solver

The program `vib.py`[19] store the complete solution $u^0, u^1, \ldots, u^{N_t}$ in memory, which is convenient for later plotting. Make a memory minimizing version of this program where only the last three u^{n+1}, u^n, and u^{n-1} values are stored in memory. Write each computed (t_{n+1}, u^{n+1}) pair to file. Visualize the data in the file (a cool solution is to read one line at a time and plot the u value using the line-by-line plotter in the `visualize_front_ascii` function - this technique makes it trivial to visualize very long time simulations). Filename: `vib_memsave`.

[19] `http://tinyurl.com/nm5587k/vib/vib.py`

Exercise 14: Implement the solver via classes

Reimplement the `vib.py` program using a class `Problem` to hold all the physical parameters of the problem, a class `Solver` to hold the numerical parameters and compute the solution, and a class `Visualizer` to display the solution.

Hint. Use the ideas and examples from Section **??** and **??**in [1]. More specifically, make a superclass `Problem` for holding the scalar physical parameters of a problem and let subclasses implement the $s(u)$ and $F(t)$ functions as methods. Try to call up as much existing functionality in `vib.py` as possible.
Filename: `vib_class`.

Exercise 15: Interpret $[D_tD_tu]^n$ as a forward-backward difference

Show that the difference $[D_tD_tu]^n$ is equal to $[D_t^+D_t^-u]^n$ and $D_t^-D_t^+u]^n$. That is, instead of applying a centered difference twice one can alternatively apply a mixture forward and backward differences. Filename: `vib_DtDt_fw_bw`.

Exercise 16: Use a backward difference for the damping term

As an alternative to discretizing the damping terms $\beta u'$ and $\beta|u'|u'$ by centered differences, we may apply backward differences:

$$
\begin{aligned}
[u']^n &\approx [D_t^-u]^n, \\
[|u'|u']^n &\approx [|D_t^-u|D_t^-u]^n = |[D_t^-u]^n|[D_t^-u]^n\,.
\end{aligned}
$$

The advantage of the backward difference is that the damping term is evaluated using known values u^n and u^{n-1} only. Extend the `vib.py`[20] code with a scheme based on using backward differences in the damping terms. Add statements to compare the original approach with centered difference and the new idea launched in this exercise. Perform numerical experiments to investigate how much accuracy that is lost by using the backward differences. Filename: `vib_gen_bwdamping`.

Exercise 17: Analysis of the Euler-Cromer scheme

The Euler-Cromer scheme for the model problem $u'' + \omega^2 u = 0$, $u(0) = I$, $u'(0) = 0$, is given in (53)-(52). Find the exact discrete solutions of this scheme and show that the solution for u^n coincides with that found in Section 4.

Hint. Use an "ansatz" $u^n = I\exp(i\tilde{\omega}\Delta t\, n)$ and $v^n = qu^n$, where $\tilde{\omega}$ and q are unknown parameters. The following formula is handy:

$$
\boldsymbol{e}^{i\tilde{\omega}\Delta t} + e^{i\tilde{\omega}(-\Delta t)} - 2 = 2\left(\cosh(i\tilde{\omega}\Delta t) - 1\right) = -4\sin^2(\frac{\tilde{\omega}\Delta t}{2})\,.
$$

[20] `http://tinyurl.com/nm5587k/vib/vib.py`

10 Applications of vibration models

The following text derives some of the most well-known physical problems that lead to second-order ODE models of the type addressed in this document. We consider a simple spring-mass system; thereafter extended with nonlinear spring, damping, and external excitation; a spring-mass system with sliding friction; a simple and a physical (classical) pendulum; and an elastic pendulum.

10.1 Oscillating mass attached to a spring

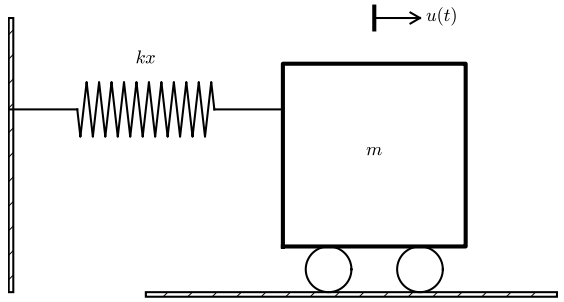


Figure 12: Simple oscillating mass.

The most fundamental mechanical vibration system is depicted in Figure 12. A body with mass m is attached to a spring and can move horizontally without friction (in the wheels). The position of the body is given by the vector $\boldsymbol{r}(t) = u(t)\boldsymbol{i}$, where $\boldsymbol{i}$ is a unit vector in x direction. There is only one force acting on the body: a spring force $\boldsymbol{F}_s = -ku\boldsymbol{i}$, where k is a constant. The point $x = 0$, where $u = 0$, must therefore correspond to the body's position where the spring is neither extended nor compressed, so the force vanishes.

The basic physical principle that governs the motion of the body is Newton's second law of motion: $\boldsymbol{F} = m\boldsymbol{a}$, where $\boldsymbol{F}$ is the sum of forces on the body, m is its mass, and $\boldsymbol{a} = \ddot{\boldsymbol{r}}$ is the acceleration. We use the dot for differentiation with respect to time, which is usual in mechanics. Newton's second law simplifies here to $-\boldsymbol{F}_s = m\ddot{u}\boldsymbol{i}$, which translates to

$$
-ku = m\ddot{u}\,.
$$

Two initial conditions are needed: $u(0) = I$, $\dot{u}(0) = V$. The ODE problem is normally written as

$$
m\ddot{u} + ku = 0, \quad u(0) = I,\ \dot{u}(0) = V\,. \tag{81}
$$

It is not uncommon to divide by m and introduce the frequency $\omega = \sqrt{k/m}$:

$$
\ddot{u} + \omega^2 u = 0, \quad u(0) = I,\ \dot{u}(0) = V\,. \tag{82}
$$

This is the model problem in the first part of this chapter, with the small difference that we write the time derivative of u with a dot above, while we used u' and u'' in previous parts of the document.

Since only one scalar mathematical quantity, $u(t)$, describes the complete motion, we say that the mechanical system has one degree of freedom (DOF).

Scaling. For numerical simulations it is very convenient to scale (82) and thereby get rid of the problem of finding relevant values for all the parameters m, k, I, and V. Since the amplitude of the oscillations are dictated by I and V (or more precisely, V/ω), we scale u by I (or $V/omega$ if $I = 0$):

$$\bar{u} = \frac{u}{I}, \quad \bar{t} = \frac{t}{t_c}.$$

The time scale t_c is normally chosen as the inverse period $2\pi/\omega$ or angular frequency $1/\omega$, most often as $t_c = 1/\omega$. Inserting the dimensionless quantities $\bar{u}$ and $\bar{t}$ in (82) results in the scaled problem

$$\frac{d^2\bar{u}}{d\bar{t}^2} + \bar{u} = 0, \quad \bar{u}(0) = 1, \ \frac{\bar{u}}{\bar{t}}(0) = \beta = \frac{V}{I\omega},$$

where β is a dimensionless number. Any motion that starts from rest ($V = 0$) is free of parameters in the scaled model!

The physics. The typical physics of the system in Figure 12 can be described as follows. Initially, we displace the body to some position I, say at rest ($V = 0$). After releasing the body, the spring, which is extended, will act with a force $-kI\boldsymbol{i}$ and pull the body to the left. This force causes an acceleration and therefore increases velocity. The body passes the point $x = 0$, where $u = 0$, and the spring will then be compressed and act with a force $kx\boldsymbol{i}$ against the motion and cause retardation. At some point, the motion stops and the velocity is zero, before the spring force $kx\boldsymbol{i}$ accelerates the body in positive direction. The result is that the body accelerates back and forth. As long as there is no friction forces to damp the motion, the oscillations will continue forever.

10.2 General mechanical vibrating system

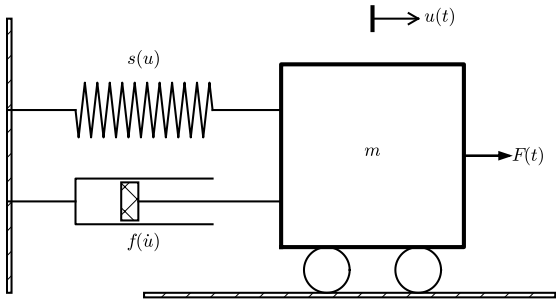


Figure 13: General oscillating system.

The mechanical system in Figure 12 can easily be extended to the more general system in Figure 13, where the body is attached to a spring and a dashpot, and also subject to an environmental force $F(t)\boldsymbol{i}$. The system has still only one degree of freedom since the body can only move back and forth parallel to the x axis. The spring force was linear, $\boldsymbol{F}_s = -ku\boldsymbol{i}$, in

Section 10.1, but in more general cases it can depend nonlinearly on the position. We therefore set $\boldsymbol{F}_s = s(u)\boldsymbol{i}$. The dashpot, which acts as a damper, results in a force $\boldsymbol{F}_d$ that depends on the body's velocity $\dot{u}$ and that always acts against the motion. The mathematical model of the force is written $\boldsymbol{F}_d = f(\dot{u})\boldsymbol{i}$. A positive $\dot{u}$ must result in a force acting in the positive x direction. Finally, we have the external environmental force $\boldsymbol{F}_e = F(t)\boldsymbol{i}$.

Newton's second law of motion now involves three forces:

$$F(t)\boldsymbol{i} + f(\dot{u})ii - s(u)\boldsymbol{i} = m\ddot{u}\boldsymbol{i}.$$

The common mathematical form of the ODE problem is

$$m\ddot{u} + f(\dot{u}) + s(u) = F(t), \quad u(0) = I, \ \dot{u}(0) = V. \tag{83}$$

This is the generalized problem treated in the last part of the present chapter, but with prime denoting the derivative instead of the dot.

The most common models for the spring and dashpot are linear: $f(\dot{u}) = b\dot{u}$ with a constant $b \geq 0$, and $s(u) = ku$ for a constant k.

Scaling. A specific scaling requires specific choices of f, s, and F. Suppose we have

$$f(\dot{u}) = b|\dot{u}|\dot{u}, \quad s(u) = ku, \quad F(t) = A\sin(\phi t).$$

We introduce dimensionless variables as usual, $\bar{u} = u/u_c$ and $\bar{t} = t/t_c$. The scale u_c depends both on the initial conditions and F, but as time grows, the effect of the initial conditions die out and F will drive the motion. Inserting $\bar{u}$ and $\bar{t}$ in the ODE gives

$$m\frac{u_c}{t_c^2}\frac{d^2\bar{u}}{d\bar{t}^2} + b\frac{u_c^2}{t_c^2}\left|\frac{d\bar{u}}{d\bar{t}}\right|\frac{d\bar{u}}{d\bar{t}} + ku_c\bar{u} = A\sin(\phi t_c\bar{t}).$$

We divide by u_c/t_c^2 and demand the coefficients of the $\bar{u}$ and the forcing term from $F(t)$ to have unit coefficients. This leads to the scales

$$t_c = \sqrt{\frac{m}{k}}, \quad u_c = \frac{A}{k}.$$

The scaled ODE becomes

$$\frac{d^2\bar{u}}{d\bar{t}^2} + 2\beta\left|\frac{d\bar{u}}{d\bar{t}}\right|\frac{d\bar{u}}{d\bar{t}} + \bar{u} = \sin(\gamma\bar{t}), \tag{84}$$

where there are two dimensionless numbers:

$$\beta = \frac{Ab}{2mk}, \quad \gamma = \phi\sqrt{\frac{m}{k}}.$$

The β number measures the size of the damping term (relative to unity) and is assumed to be small, basically because b is small. The ϕ number is the ratio of the time scale of free vibrations and the time scale of the forcing. The scaled initial conditions have two other dimensionless numbers as values:

$$\bar{u}(0) = \frac{Ik}{A}, \quad \frac{d\bar{u}}{d\bar{t}} = \frac{t_c}{u_c}V = \frac{V}{A}\sqrt{mk}.$$

10.3 A sliding mass attached to a spring

Consider a variant of the oscillating body in Section 10.1 and Figure 12: the body rests on a flat surface, and there is sliding friction between the body and the surface. Figure 14 depicts the problem.

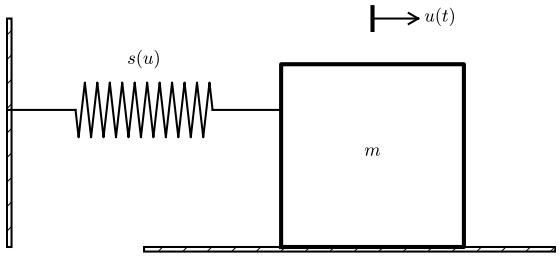


Figure 14: Sketch of a body sliding on a surface.

The body is attached to a spring with spring force $-s(u)\boldsymbol{i}$. The friction force is proportional to the normal force on the surface, $-mg\boldsymbol{j}$, and given by $-f(\dot{u})\boldsymbol{i}$, where

$$f(\dot{u}) = \begin{cases} -\mu mg, & \dot{u} < 0, \\ \mu mg, & \dot{u} > 0, \\ 0, & \dot{u} = 0 \end{cases}$$

Here, μ is a friction coefficient. With the signum function

$$\text{sign(x)} = \begin{cases} -1, & x < 0, \\ 1, & x > 0, \\ 0, & x = 0 \end{cases}$$

we can simply write $f(\dot{u}) = \mu mg\,\text{sign}(\dot{u})$ (the sign function is implemented by `numpy.sign`).

The equation of motion becomes

$$m\ddot{u} + \mu mg\text{sign}(\dot{u}) + s(u) = 0, \quad u(0) = I,\ \dot{u}(0) = V\,. \tag{85}$$

10.4 A jumping washing machine

A washing machine is placed on four springs with efficient dampers. If the machine contains just a few clothes, the circular motion of the machine induces a sinusoidal external force and the machine will jump up and down if the frequency of the external force is close to the natural frequency of the machine and its spring-damper system.

10.5 Motion of a pendulum

A classical problem in mechanics is the motion of a pendulum. We first consider a simple pendulum[21]: a small body of mass m is attached to a massless wire and can oscillate back and forth in the gravity field. Figure 15 shows a sketch of the problem.

[21] `https://en.wikipedia.org/wiki/Pendulum`

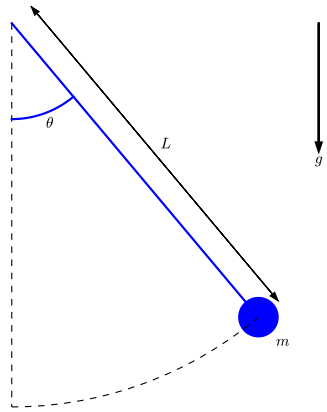


Figure 15: Sketch of a simple pendulum.

The motion is governed by Newton's 2nd law, so we need to find expressions for the forces and the acceleration. Three forces on the body are considered: an unknown force S from the wire, the gravity force mg, and an air resistance force, $\frac{1}{2}C_D\varrho A|v|v$, hereafter called the drag force, directed against the velocity of the body. Here, C_D is a drag coefficient, ϱ is the density of air, A is the cross section area of the body, and v is the velocity.

We introduce a coordinate system with polar coordinates and unit vectors $\boldsymbol{i}_r$ and $\boldsymbol{i}_\theta$ as shown in Figure 16. The position of the center of mass of the body is

$$\boldsymbol{r}(t) = x_0\boldsymbol{i} + y_0\boldsymbol{j} + L\boldsymbol{i}_r,$$

where $\boldsymbol{i}$ and $\boldsymbol{j}$ are unit vectors in the corresponding Cartesian coordinate system in the x and y directions, respectively. We have that $\boldsymbol{i}_r = \cos\theta\boldsymbol{i} + \sin\theta\boldsymbol{j}$.

The forces are now expressed as follows.

- Wire force: $-S\boldsymbol{i}_r$
- Gravity force: $-mg\boldsymbol{j} = mg(-\sin\theta\boldsymbol{i}_\theta + \cos\theta\boldsymbol{i}_r)$
- Drag force: $-\frac{1}{2}C_D\varrho A|v|v\boldsymbol{i}_\theta$

Since a positive velocity means movement in the direction of $\boldsymbol{i}_\theta$, the drag force must be directed along $-\boldsymbol{i}_\theta$.

The velocity of the body is found from $\boldsymbol{r}$:

$$\boldsymbol{v}(t) = \dot{\boldsymbol{r}}(t) = \frac{d}{d\theta}(x_0\boldsymbol{i} + y_0\boldsymbol{j} + L\boldsymbol{i}_r)\frac{d\theta}{dt} = L\dot{\theta}\boldsymbol{i}_\theta,$$

since $\frac{d}{d\theta}\boldsymbol{i}_r = \boldsymbol{i}_\theta$. It follows that $v = |\boldsymbol{v}| = L\dot{\theta}$. The acceleration is

$$\boldsymbol{a}(t) = \dot{\boldsymbol{v}}(r) = \frac{d}{dt}(L\dot{\theta}\boldsymbol{i}_\theta) = L\ddot{\theta}\boldsymbol{i}_\theta + L\dot{\theta}\frac{d\boldsymbol{i}_\theta}{d\theta}\dot{\theta} == L\ddot{\theta}\boldsymbol{i}_\theta - L\dot{\theta}^2\boldsymbol{i}_r,$$

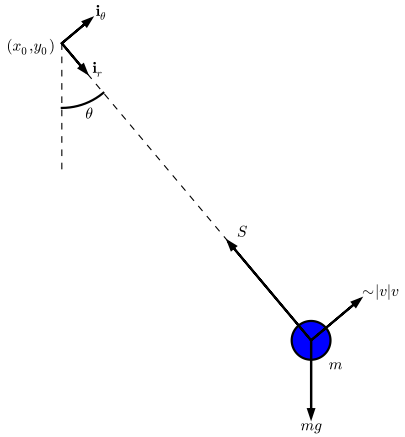


Figure 16: Forces acting on a simple pendulum.

since $\frac{d}{d\theta}\boldsymbol{i}_\theta = -\boldsymbol{i}_r$.

Newton's 2nd law of motion becomes

$$-S\boldsymbol{i}_r + mg(-\sin\theta\boldsymbol{i}_\theta + \cos\theta\boldsymbol{i}_r) - \frac{1}{2}C_D\varrho AL^2|\dot\theta|\dot\theta\boldsymbol{i}_\theta = mL\ddot\theta\dot\theta\boldsymbol{i}_\theta - L\dot\theta^2\boldsymbol{i}_r,$$

leading to two component equations

$$-S + mg\cos\theta = -L\dot\theta^2, \tag{86}$$

$$-mg\sin\theta - \frac{1}{2}C_D\varrho AL^2|\dot\theta|\dot\theta = mL\ddot\theta\,. \tag{87}$$

From (86) we get an expression for $S = mg\cos\theta + L\dot\theta^2$, and from (87) we get a differential equation for the angle $\theta(t)$. This latter equation is ordered as

$$m\ddot\theta + +\frac{1}{2}C_D\varrho AL|\dot\theta|\dot\theta + \frac{mg}{L}\sin\theta = 0\,. \tag{88}$$

Two initial conditions are needed: $\theta = \Theta$ and $\dot\theta = \Omega$. Normally, the pendulum motion is started from rest, which means $\Omega = 0$.

Equation (88) fits the general model used in (58) in Section 8 if we define $u = \theta$, $f(u') = \frac{1}{2}C_D\varrho AL|\dot\theta|\dot\theta$, $s(u) = L^{-1}mg\sin u$, and $F = 0$. If the body is a sphere with radius R, we can take $C_D = 0.4$ and $A = \pi R^2$.

The motion of a compound or physical pendulum where the wire is a rod with mass, can be modeled very similarly. The governing equation is $I\boldsymbol{a} = \boldsymbol{T}$ where I is the moment of inertia of the entire body about the point (x_0, y_0), and $\boldsymbol{T}$ is the sum of moments of the forces with respect to (x_0, y_0). The vector equation reads

$$\boldsymbol{r}\times(-S\boldsymbol{i}_r + mg(-\sin\theta\boldsymbol{i}_\theta + \cos\theta\boldsymbol{i}_r) - \frac{1}{2}C_D\varrho AL^2|\dot\theta|\dot\theta\boldsymbol{i}_\theta) = I(L\ddot\theta\dot\theta\boldsymbol{i}_\theta - L\dot\theta^2\boldsymbol{i}_r)\,.$$

The component equation in $\boldsymbol{i}_\theta$ direction gives the equation of motion for $\theta(t)$:

$$I\ddot\theta + \frac{1}{2}C_D\varrho AL^3|\dot\theta|\dot\theta + mgL\sin\theta = 0\,. \tag{89}$$

10.6 Motion of an elastic pendulum

Consider a pendulum as in Figure 15, but this time the wire is elastic. The length of the wire when it is not stretched is L_0, while $L(t)$ is the stretched length at time t during the motion.

Stretching the elastic wire a distance ΔL gives rise to a spring force $k\Delta L$ in the opposite direction of the stretching. Let $\boldsymbol{n}$ be a unit normal vector along the wire from the point $\boldsymbol{r}_0 = (x_0, y_0)$ and in the direction of $\boldsymbol{i}_\theta$, see Figure 16 for definition of (x_0, y_0) and $\boldsymbol{i}_\theta$. Obviously, we have $\boldsymbol{n} = \boldsymbol{i}_\theta$, but in this modeling of an elastic pendulum we do not need polar coordinates. Instead, it is more straightforward to develop the equation in Cartesian coordinates.

A mathematical expression for $\boldsymbol{n}$ is

$$\boldsymbol{n} = \frac{\boldsymbol{r} - \boldsymbol{r}_0}{L(t)},$$

where $L(t) = ||\boldsymbol{r} - \boldsymbol{r}_0||$ is the current length of the elastic wire. The position vector $\boldsymbol{r}$ in Cartesian coordinates reads $\boldsymbol{r}(t) = x(t)\boldsymbol{i} + y(t)\boldsymbol{j}$, where $\boldsymbol{i}$ and $\boldsymbol{j}$ are unit vectors in the x and y directions, respectively. It is convenient to introduce the Cartesian components n_x and n_y of the normal vector:

$$\boldsymbol{n} = \frac{\boldsymbol{r} - \boldsymbol{r}_0}{L(t)} = \frac{x(t) - x_0}{L(t)}\boldsymbol{i} + \frac{y(t) - y_0}{L(t)}\boldsymbol{j} = n_x\boldsymbol{i} + n_y\boldsymbol{j}\,.$$

The stretch ΔL in the wire is

$$\Delta t = L(t) - L_0\,.$$

The force in the wire is then $-S\boldsymbol{n} = -k\Delta L\boldsymbol{n}$.

The other forces are the gravity and the air resistance, just as in Figure 16. The main difference is that we have a *model* for S in terms of the motion (as soon as we have expressed ΔL by $\boldsymbol{r}$). For simplicity, we drop the air resistance term (but Exercise 22 asks you to include it).

Newton's second law of motion applied to the body now results in

$$m\ddot{\boldsymbol{r}} = -k(L - L_0)\boldsymbol{n} - mg\boldsymbol{j} \tag{90}$$

The two components of (90) are

$$\ddot x = -\frac{k}{m}(L - L_0)n_x, \tag{91}$$

$$\tag{92}$$

$$\ddot y = -\frac{k}{m}(L - L_0)n_y - g\,. \tag{93}$$

Remarks about an elastic vs a non-elastic pendulum. Note that the derivation of the ODEs for an elastic pendulum is more straightforward than for a classical, non-elastic pendulum, since we avoid the details with polar coordinates, but instead work with Newton's second law directly in Cartesian coordinates. The reason why we can do this is that the elastic pendulum undergoes a general two-dimensional motion where all the forces are known or expressed as functions of $x(t)$ and $y(t)$, such that we get two ordinary differential equations. The motion of the non-elastic pendulum, on the other hand, is constrained: the body has to move along a circular path, and the force S in the wire is unknown.

The non-elastic pendulum therefore leads to a *differential-algebraic* equation, i.e., ODEs for $x(t)$ and $y(t)$ combined with an extra constraint $(x-x_0)^2 + (y-y_0)^2 = L^2$ ensuring that the motion takes place along a circular path. The extra constraint (equation) is compensated by an extra unknown force $-S\boldsymbol{n}$. Differential-algebraic equations are normally hard to solve, especially with pen and paper. Fortunately, for the non-elastic pendulum we can do a trick: in polar coordinates the unknown force S appears only in the radial component of Newton's second law, while the unknown degree of freedom for describing the motion, the angle $\theta(t)$, is completely governed by the asimuthal component. This allows us to decouple the unknowns S and θ. But this is a kind of trick and not a widely applicable method. With an elastic pendulum we use straightforward reasoning with Newton's 2nd law and arrive at a standard ODE problem that (after scaling) is easy solve on a computer.

Initial conditions. What is the initial position of the body? We imagine that first the pendulum hangs in equilibrium in its vertical position, and then it is displaced an angle Θ. The equilibrium position is governed by the ODEs with the accelerations set to zero. The x component leads to $x(t) = x_0$, while the y component gives

$$0 = -\frac{k}{m}(L - L_0)n_y - g = \frac{k}{m}(L(0) - L_0) - g \quad \Rightarrow \quad L(0) = L_0 + mg/k,$$

since $n_y = -11$ in this position. The corresponding y value is then from $n_y = -1$:

$$y(t) = y_0 - L(0) = y_0 - (L_0 + mg/k)\,.$$

Let us now choose (x_0, y_0) such that the body is at the origin in the equilibrium position:

$$x_0 = 0, \quad y_0 = L_0 + mg/k\,.$$

Displacing the body an angle Θ to the right leads to the initial position

$$x(0) = (L_0 + mg/k)\sin\Theta, \quad y(0) = (L_0 + mg/k)(1 - \cos\Theta)\,.$$

The initial velocities can be set to zero: $x'(0) = y'(0) = 0$.

The complete ODE problem. We can summarize all the equations as follows:

$$\begin{aligned}
\ddot{x} &= -\frac{k}{m}(L - L_0)n_x,\\
\ddot{y} &= -\frac{k}{m}(L - L_0)n_y - g,\\
L &= \sqrt{(x-x_0)^2 + (y-y_0)^2},\\
n_x &= \frac{x - x_0}{L},\\
n_y &= \frac{y - y_0}{L},\\
x(0) &= (L_0 + mg/k)\sin\Theta,\\
x'(0) &= 0,\\
y(0) &= (L_0 + mg/k)(1 - \cos\Theta),\\
y'(0) &= 0\,.
\end{aligned}$$

We insert n_x and n_y in the ODEs:

$$\ddot{x} = -\frac{k}{m}\left(1 - \frac{L_0}{L}\right)(x - x_0), \tag{94}$$

$$\ddot{y} = -\frac{k}{m}\left(1 - \frac{L_0}{L}\right)(y - y_0) - g, \tag{95}$$

$$L = \sqrt{(x-x_0)^2 + (y-y_0)^2}, \tag{96}$$

$$x(0) = (L_0 + mg/k)\sin\Theta, \tag{97}$$

$$x'(0) = 0, \tag{98}$$

$$y(0) = (L_0 + mg/k)(1 - \cos\Theta), \tag{99}$$

$$y'(0) = 0\,. \tag{100}$$

Scaling. The elastic pendulum model can be used to study both an elastic pendulum and a classic, non-elastic pendulum. The latter problem is obtained by letting $k \to \infty$. Unfortunately, a serious problem with the ODEs (94)-(95) is that for large k, we have a very large factor k/m multiplied by a very small number $1 - L_0/L$, since for large k, $L \approx L_0$ (very small deformations of the wire). The product is subject to significant round-off errors for many relevant physical values of the parameters. To circumvent the problem, we introduce a scaling. This will also remove physical parameters from the problem such that we end up with only one dimensionless parameter, closely related to the elasticity of the wire. Simulations can then be done by setting just this dimensionless parameter.

The characteristic length can be taken such that in equilibrium, the scaled length is unity, i.e., the characteristic length is $L_0 + mg/k$:

$$\bar{x} = \frac{x}{L_0 + mg/k}, \quad \bar{y} = \frac{y}{L_0 + mg/k}\,.$$

We must then also work with the scaled length $\bar{L} = L/(L_0 + mg/k)$.

Introducing $\bar{t} = t/t_c$, where t_c is a characteristic time we have to decide upon later, one gets

$$
\begin{aligned}
\frac{d^2\bar{x}}{d\bar{t}^2} &= -t_c^2\frac{k}{m}\left(1-\frac{L_0}{L_0+mg/k}\frac{1}{\bar{L}}\right)\bar{x},\\
\frac{d^2\bar{y}}{d\bar{t}^2} &= -t_c^2\frac{k}{m}\left(1-\frac{L_0}{L_0+mg/k}\frac{1}{\bar{L}}\right)(\bar{y}-1) - t_c^2\frac{g}{L_0+mg/k},\\
\bar{L} &= \sqrt{\bar{x}^2+(\bar{y}-1)^2},\\
\bar{x}(0) &= \sin\Theta,\\
\bar{x}'(0) &= 0,\\
\bar{y}(0) &= 1-\cos\Theta,\\
\bar{y}'(0) &= 0\,.
\end{aligned}
$$

For a non-elastic pendulum with small angles, we know that the frequency of the oscillations are $\omega = \sqrt{L/g}$. It is therefore natural to choose a similar expression here, either the length in the equilibrium position,

$$
t_c^2 = \frac{L_0+mg/k}{g}\,.
$$

or simply the unstretched length,

$$
t_c^2 = \frac{L_0}{g}\,.
$$

These quantities are not very different (since the elastic model is valid only for quite small elongations), so we take the latter as it is the simplest one.

The ODEs become

$$
\begin{aligned}
\frac{d^2\bar{x}}{d\bar{t}^2} &= -\frac{L_0k}{mg}\left(1-\frac{L_0}{L_0+mg/k}\frac{1}{\bar{L}}\right)\bar{x},\\
\frac{d^2\bar{y}}{d\bar{t}^2} &= -\frac{L_0k}{mg}\left(1-\frac{L_0}{L_0+mg/k}\frac{1}{\bar{L}}\right)(\bar{y}-1) - \frac{L_0}{L_0+mg/k},\\
\bar{L} &= \sqrt{\bar{x}^2+(\bar{y}-1)^2}\,.
\end{aligned}
$$

We can now identify a dimensionless number

$$
\beta = \frac{L_0}{L_0+mg/k} = \frac{1}{1+\frac{mg}{L_0k}},
$$

which is the ratio of the unstretched length and the stretched length in equilibrium. The non-elastic pendulum will have $\beta = 1$ ($k\to\infty$). With β the ODEs read

$$
\begin{aligned}
\frac{d^2\bar{x}}{d\bar{t}^2} &= -\frac{\beta}{1-\beta}\left(1-\frac{\beta}{\bar{L}}\right)\bar{x}, && (101)\\
\frac{d^2\bar{y}}{d\bar{t}^2} &= -\frac{\beta}{1-\beta}\left(1-\frac{\beta}{\bar{L}}\right)(\bar{y}-1) - \beta, && (102)\\
\bar{L} &= \sqrt{\bar{x}^2+(\bar{y}-1)^2}, && (103)\\
\bar{x}(0) &= (1+\epsilon)\sin\Theta, && (104)\\
\frac{d\bar{x}}{d\bar{t}}(0) &= 0, && (105)\\
\bar{y}(0) &= 1-(1+\epsilon)\cos\Theta, && (106)\\
\frac{d\bar{y}}{d\bar{t}}(0) &= 0, && (107)
\end{aligned}
$$

We have here added a parameter ϵ, which is an additional downward stretch of the wire at $t=0$. This parameter makes it possible to do a desired test: vertical oscillations of the pendulum. Without ϵ, starting the motion from $(0,0)$ with zero velocity will result in $x=y=0$ for all times (also a good test!), but with an initial stretch so the body's position is $(0,\epsilon)$, we will have oscillatory vertical motion with amplitude ϵ (see Exercise 21).

Remark on the non-elastic limit. We immediately see that as $k\to\infty$ (i.e., we obtain a non-elastic pendulum), $\beta\to 1$, $\bar{L}\to 1$, and we have very small values $1-\beta\bar{L}^{-1}$ divided by very small values $1-\beta$ in the ODEs. However, it turns out that we can set β very close to one and obtain a path of the body that within the visual accuracy of a plot does not show any elastic oscillations. (Should the division of very small values become a problem, one can study the limit by L'Hospital's rule:

$$
\lim_{\beta\to 1}\frac{1-\beta\bar{L}^{-1}}{1-\beta} = \frac{1}{\bar{L}},
$$

and use the limit $\bar{L}^{-1}$ in the ODEs for β values very close to 1.)

10.7 Bouncing ball

A bouncing ball is a body in free vertically fall until it impacts the ground. During the impact, some kinetic energy is lost, and a new motion upwards with reduced velocity starts. At some point the velocity close to the ground is so small that the ball is considered to be finally at rest.

The motion of the ball falling in air is governed by Newton's second law $F=ma$, where a is the acceleration of the body, m is the mass, and F is the sum of all forces. Here, we neglect the air resistance so that gravity $-mg$ is the only force. The height of the ball is denoted by h and v is the velocity. The relations between h, v, and a,

$$
h'(t) = v(t), \quad v'(t) = a(t),
$$

combined with Newton's second law gives the ODE model

$$
h''(t) = -g, \tag{108}
$$

or expressed alternatively as a system of first-order equations:

$$v'(t) = -g, \tag{109}$$
$$h'(t) = v(t)\,. \tag{110}$$

These equations govern the motion as long as the ball is away from the ground by a small distance $\epsilon_h > 0$. When $h < \epsilon_h$, we have two cases.

1. The ball impacts the ground, recognized by a sufficiently large negative velocity ($v < -\epsilon_v$). The velocity then changes sign and is reduced by a factor C_R, known as the coefficient of restitution[22]. For plotting purposes, one may set $h = 0$.
2. The motion stops, recognized by a sufficiently small velocity ($|v| < \epsilon_v$) close to the ground.

10.8 Electric circuits

Although the term "mechanical vibrations" is used in the present document, we must mention that the same type of equations arise when modeling electric circuits. The current $I(t)$ in a circuit with an inductor with inductance L, a capacitor with capacitance C, and overall resistance R, is governed by

$$\ddot{I} + \frac{R}{L}\dot{I} + \frac{1}{LC}I = \dot{V}(t), \tag{111}$$

where $V(t)$ is the voltage source powering the circuit. This equation has the same form as the general model considered in Section refrefvib:model2 if we set $u = I$, $f(u' = bu'$ and define $b = R/L$, $s(u) = L^{-1}C^{-1}u$, and $F(t) = \dot{V}(t)$.

11 Exercises

Exercise 18: Simulate resonance

We consider the scaled ODE model (84) from Section 10.2. After scaling, the amplitude of u will have a size about unity as time grows and the effect of the initial conditions die out due to damping. However, as $\gamma \to 1$, the amplitude of u increases, especially if β is small. This effect is called *resonance*. The purpose of this exercise is to explore resonance.

a) Figure out how the `solver` function in `vib.py` can be called for the scaled ODE (84).

b) Run $\gamma = 5, 1.5, 1.1, 1$ for $\beta = 0.005, 0.05, 0.2$. For each β value, present an image with plots of $u(t)$ for the four γ values.
Filename: `resonance`.

Exercise 19: Simulate oscillations of a sliding box

Consider a sliding box on a flat surface as modeled in Section 10.3. As spring force we choose the nonlinear formula

$$s(u) = \frac{k}{\alpha}\tanh(\alpha u) = ku + \frac{1}{3}\alpha^2 ku^3 + \frac{2}{15}\alpha^4 ku^5 + \mathcal{O}(u^6)\,.$$

[22] `http://en.wikipedia.org/wiki/Coefficient_of_restitution`

a) Plot $g(u) = \alpha^{-1}\tanh(\alpha u)$ for various values of α. Assume $u \in [-1, 1]$.

b) Scale the equations using I as scale for u and m/k as time scale.

c) Implement the scaled model in b). Run it for some values of the dimensionless parameters.
Filename: `sliding_box`.

Exercise 20: Simulate a bouncing ball

Section 10.7 presents a model for a bouncing ball. Choose one of the two ODE formulation, (108) or (109)-(110), and simulate the motion of a bouncing ball. Plot $h(t)$. Think about how to plot $v(t)$.

Hint. A naive implementation may get stuck in repeated impacts for large time step sizes. To avoid this situation, one can introduce a state variable that holds the mode of the motion: free fall, impact, or rest. Two consecutive impacts imply that the motion has stopped.
Filename: `bouncing_ball`.

Exercise 21: Simulate an elastic pendulum

Section 10.6 describes a model for an elastic pendulum, resulting in a system of two ODEs. The purpose of this exercise is to implement the scaled model, test the software, and generalize the model.

a) Write a function `simulate` that can simulate an elastic pendulum using the scaled model. The function should have the following arguments:

```
def simulate(
    beta=0.9,                # dimensionless parameter
    Theta=30,                # initial angle in degrees
    epsilon=0,               # initial stretch of wire
    num_periods=6,           # simulate for num_periods
    time_steps_per_period=60, # time step resolution
    plot=True,               # make plots or not
    ):
```

To set the total simulation time and the time step, we use our knowledge of the scaled, classical, non-elastic pendulum: $u'' + u = 0$, with solution $u = \Theta\cos\bar{t}$. The period of these oscillations is $P = 2\pi$ and the frequency is unity. The time for simulation is taken as `num_periods` times P. The time step is set as P divided by `time_steps_per_period`.

The `simulate` function should return the arrays of x, y, θ, and t, where $\theta = \tan^{-1}(x/(1-y))$ is the angular displacement of the elastic pendulum corresponding to the position (x, y).

If `plot` is `True`, make a plot of $\bar{y}(\bar{t})$ versus $\bar{x}(\bar{t})$, i.e., the physical motion of the mass at $(\bar{x}, \bar{y})$. Use the equal aspect ratio on the axis such that we get a physically correct picture of the motion. Also make a plot of $\theta(\bar{t})$, where θ is measured in degrees. If $\Theta < 10$ degrees, add a plot that compares the solutions of the scaled, classical, non-elastic pendulum and the elastic pendulum ($\theta(t)$).

Although the mathematics here employs a bar over scaled quantities, the code should feature plain names `x` for $\bar{x}$, `y` for $\bar{y}$, and `t` for $\bar{t}$ (rather than `x_bar`, etc.). These variable names make the code easier to read and compare with the mathematics.

Hint 1. Equal aspect ratio is set by `plt.gca().set_aspect('equal')` in Matplotlib (`import matplotlib.pyplot as plt`) and by `plot(..., daspect=[1,1,1], daspectmode='equal')` in SciTools (`import scitools.std as plt`).

Hint 2. If you want to use Odespy to solve the equations, order the ODEs like $\dot{\bar{x}}, \bar{x}, \dot{\bar{y}}, \bar{y}$ such that the Euler-Cromer scheme can (also) be used (`odespy.EulerCromer`).

b) Write a test function for testing that $\Theta = 0$ and $\epsilon = 0$ gives $x = y = 0$ for all times.

c) Write another test function for checking that the pure vertical motion of the elastic pendulum is correct. Start with simplifying the ODEs for pure vertical motion and show that $\bar{y}(\bar{t})$ fulfills a vibration equation with frequency $\sqrt{\beta/(1-\beta)}$. Set up the exact solution.

Write a test function that uses this special case to verify the `simulate` function. There will be numerical approximation errors present in the results from `simulate` so you have to believe in correct results and set a (low) tolerance that corresponds to the computed maximum error. Use a small Δt to obtain a small numerical approximation error.

d) Make a function `demo(beta, Theta)` for simulating an elastic pendulum with a given β parameter and initial angle Θ. Use 600 time steps per period to get every accurate results, and simulate for 3 periods.
Filename: `elastic_pendulum`.

Exercise 22: Simulate an elastic pendulum with air resistance

This is a continuation Exercise 22. Air resistance on the body with mass m can be modeled by the force $-\frac{1}{2}\varrho C_D A|\boldsymbol{v}|\boldsymbol{v}$, where C_D is a drag coefficient (0.2 for a sphere), ϱ is the density of air (1.2 kg m^{-3}), A is the cross section area ($A = \pi R^2$ for a sphere, where R is the radius), and $\boldsymbol{v}$ is the velocity of the body. Include air resistance in the original model, scale the model, write a function `simulate_drag` that is a copy of the `simulate` function from Exercise 22, but with the new ODEs included, and show plots of how air resistance influences the motion.
Filename: `elastic_pendulum_drag`.

Remarks. Test functions are challenging to construct for the problem with air resistance. You can reuse the tests from Exercise 22 for `simulate_drag`, but these tests does not verify the new terms arising from air resistance.

References

[1] H. P. Langtangen. *Finite Difference Computing with Exponential Decay Models*. 2015. *http://tinyurl.com/nclmcng/doc/pub/book*.

Index